1991

Efficient Data Structures and Algorithms for Scientific Computations.

Soon Cheol Park
Louisiana State University and Agricultural & Mechanical College

Order Number 9219565

Efficient data structures and algorithms for scientific computations

Park, Soon Cheol, Ph.D.

The Louisiana State University and Agricultural and Mechanical Col., 1991

U·M·I
300 N. Zeeb Rd.
Ann Arbor, MI 48106

Efficient Data Structures and Algorithms for Scientific Computations

A Dissertation

Submitted to the Graduate Faculty of the
Louisiana State University and
Agricultural and Mechanical College
in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy

in

The Department of Computer Science

by

Soon Cheol Park
B.Eng., Inha University of Korea, 1979
M.Eng., Inha University of Korea, 1982
M.S., Hampton University, USA, 1986
December, 1991

ACKNOWLEDGMENTS

The author wishes to express the most sincere gratitude to his advisor Professor J. P. Draayer for his guidance and encouragement during the course of this research and for the criticism of the manuscript. He is also indebted to Professor S.-Q. Zheng for his advice and help in the theoretical work. He further would like to thank Professors J. E. Tohline, S. S. Iyengar and D. H. Kraft for their assistance throughout his graduate study period.

In addition, many enjoyable discussions with the members of the Korean study group in the Department of Computer Science proved valuable, and the author's appreciation is extended to Mr. Y. Cho, Mr. W. Jung, Mr. K. Kwon, Mr. S. Lee and Mr. J. Lim. The good times shared with Dr. P. Rochford, Mr. C. Bahri, Miss J. Escher and Mr. H. Naqvi will be remembered warmly.

The author wishes to thank his parents and his wife for their love and selfless support.

Finally, the author acknowledge the support from both the Department of Physics and Astronomy and the Department of Computer Science.

TABLE OF CONTENTS

LIST OF TABLES

Table **Page**

LIST OF FIGURES

ABSTRACT

Large-scale numerically intensive scientific applications can require tremendous amounts of computer time and space. Two general methods are presented for reducing the computer resources required in scientific computing. The first is a numerical database system which is built on a space and time optimal data structure called a weighted search tree and that allows for the storage and retrieval of valuable intermediate information so costly redundant calculations can be avoided. The second is a matrix algorithm based on a new space optimal representation of sparse matrices that for typical scientific applications can be expected to dramatically decrease the cost of multiplying sparse matrices. Codes and tests for each are given. Both methods can be implemented in a broad range of large-scale scientific applications.

CHAPTER 1
INTRODUCTION

Finding efficient means for performing numerical calculations on a computer is a goal which computer scientists are always trying to achieve. This objective acquires greater importance when implementing algorithms which are used for large-scale numerical calculations, as the cost of conducting such calculations can not only be substantial but also prohibitive, and may in some cases be impossible to perform regardless of the cost with existing computers. One means of accomplishing this goal is to reduce the number of redundant calculations which are done, while another is to find efficient algorithms and data structures for solving problems.

In large-scale numerically intensive scientific applications it is quite common that the same computation needs to be repeated many times. This usually occurs because the pregeneration of all required information is either impractical or impossible. There is therefore an opportunity to save by reusing results that are generated on-the-fly. An example from quantum physics is the calculation of overlap integrals required for the evaluation of matrix elements in a many-body hamiltonian theory [Neg 88]. While regenerating results is wasteful of CPU time, memory limitations frequently prohibit the saving of all the intermediate results. Thus, maintaining a numerical database of moderate size that keeps the most frequently used intermediate results, which can be accessed efficiently so that the redundant calculations are reduced, is a general approach that saves computing resources.

A numerical database [Che 84, Dat 86, Kor 86, Ull 89] can be characterized by the variables used to compute the information associated with each node, and the operations for accessing and maintaining the stored results. These variables can therefore be treated as keys. The research problem considered here is as follows: given a programming environment that does not allow for dynamic storage allocation and recursive procedure/function calls [Mac 83], design a data structure which supports search, insert and delete operations such that (i) the data items to be deleted are selected using a user defined priority; (ii) all on-line search, insert, and delete operations require a minimum amount of time; and (iii) the data structure requires the minimal possible space.

It should be clear that the availability and implementation of such a database can reduce frequent redundant calculations encountered in large-scale scientific applications and, as well, in non-numerical applications. A data structure called a "weighted search tree" (*wst*) was developed for this purpose. This required the development of certain component features: an iterative AVL tree delete algorithm, a combined AVL tree [Ade 62, Aho 74, Fos 73, Kro 79, Rei 83, Wir 76] and heap [Hel 86, Lip 86] data structure so a priority strategy could be incorporated, and a multilist representation [Hor 83b] of the AVL tree so a word of space for each node of the AVL tree could be saved. This resulting data structure is a user friendly algorithm which is optimal with respect to saving both time and space computing resources.

Developing an efficient means of storing and retrieving information used in the intermediate stages of a large-scale scientific calculation is one approach to making computationally intensive calculations feasible. However, even with

the gains that can be realized in this way; there still remain many applications which demand computing power that are many orders of magnitude beyond that available with conventional computers. Developing efficient algorithms for a large-scale scientific calculation can help overcome the problem. For example, the multiplication of two $n \times n$ sparse matrices has $O(nt_1 + nt_2)$ time complexity and needs $3 \times (t_1 + 1)$ and $3 \times (t_2 + 1)$ words of space using a specially designed algorithm [Hor 83], where t_1 and t_2 are the number of nonzero elements in each matrix respectively, while the standard way takes $O(n^3)$ and needs $2 \times n^2$ words of space. When n is very large, more than 1000 for example, the standard approach is impractical with conventional computers since it needs several million words of memory and takes too much time. However, in the sparse matrix case with $n >> t_1, t_2$, time and space for the multiplication of two matrices can be reduced significantly by using the special algorithm referred to above.

In fact, many large-scale scientific computations involve a large sparse matrix problem, including eigenvalue analyses [Gol 89, Parl 80], least square problems [Gol 89, Net 89], solutions to large sparse systems of linear equations [Cul 85], etcetera. For the multiplication of sparse matrices, it is desirable to design an algorithm whose time and space complexities are sensitive to the number of elements in the operand matrices. Horowitz and Sahni give a space-efficient representation of sparse matrices in which only nonzero elements of a matrix are taken into consideration. Each nonzero element is represented by a 3-tuple of the form (i, j, v), where i and j are the row and column numbers of the element, respectively, and v is the value of the element. A sparse matrix is stored in memory as a linear list of nonzero

elements in row-major order. With this representation, multiplying two $n \times n$ matrices can be carried out in $O(n\ (t_1 + t_2))$ time, where t_1 and t_2 are the numbers of nonzero elements in the two operand matrices, respectively.

A modified data structure for sparse matrices will be introduced in this dissertation. A matrix multiplication algorithm based on this new data structure will also be given. Our analysis shows that this new data structure and algorithm are more time and space efficient than existing methods if the sparse matrices contain a considerable number of segments consisting of adjacent nonzero elements in rows and/or columns. The improved performance of this algorithm is demonstrated by experiments. It is important to note that for scientific applications, even a factor of two improvement in the time efficiency may reduce the computation time by several hours or even days, and may make a significant difference in the feasibility of running a code at reasonable cost. Since the sparse matrices arising in scientific applications tend to be highly structured, such as band and triangular matrices, our new matrix multiplication algorithm can be implemented as a powerful tool for efficiently solving many important time consuming matrix problems.

In Chapter 2 the structure of the *wst* is described. Then in Chapter 3 the *wst* is implemented in developing a numerical database system that functions like a file directory system on a disk. In Chapter 4, a new efficient algorithm for the multiplication of sparse matrices is introduced and results are given which show how much time and space its implementation can save in typical applications. Finally, in Chapter 5, a summary of the performance advantages that can be achieved with the *wst* database system and new sparse matrix algorithm are presented.

CHAPTER 2
WEIGHTED SEARCH TREE

One way to make scientific computations more efficient is to reduce the number of redundant calculations that are done. A numerical database can be used to accomplish this objective. In this chapter, we will introduce a data structure, called a weighted search tree (*wst*), which can be used to build numerical databases. The *wst* is the combination of a priority queue (heap) and a height balanced binary tree. Information stored in a *wst* is assigned a weight, its priority, determined through a combination of the hit frequency and a user-defined base priority that can incorporate, for example, generation time. The *wst* supports optimal on-line search, insert and delete operations with minimum space requirements. Space optimality is achieved through the use of implicit data structures. The *wst* and its associated operations provide an invaluable user-friendly building block for large-scale numerically intensive scientific applications.

2.1 Introduction

Large-scale scientific applications can require tremendous amounts of computer time and space. So despite recent advances in the development of high performance computing systems with multiple processors and massive storage capacity, designing time-space optimal algorithms remains one of the most fundamental and important aspects of scientific computing. In general, algorithm development work is problem dependent so a methodology that

solves one problem may not work for another. It is therefore particularly important to identify and develop efficient methods for algorithms that can be used as building blocks in many different applications.

In numerically intensive work it is common to find the exact same computation repeated many times. This occurs, for example, when the pregeneration of required information is either impractical or impossible. An example from quantum physics is the calculation of overlap integrals that are required for the evaluation of matrix elements in a many-body theory [Neg 88]. While regeneration is wasteful, memory limitations usually preclude one from saving all intermediate results. Thus there is a need for an efficient numerical database [Che 84, Par 89a, Par 90b, Par 90c] so the amount of time spent on redundant calculations can be minimized subject to space constraints.

Results stored in a numerical database can be characterized by the input variables used to determine their values. These variables can therefore be treated as keys. Without loss of generality, the numerical database model considered in this chapter has only one key and three operations: search, insert and delete. The logic can be extended to include multiple keys and other operations. Specifically, the problem addressed is the following: given a programming environment that does not allow for the dynamic allocation [Mac 83] of storage nor for recursive procedure/function calls, design a data structure that supports search, insert and delete operations such that (i) deleted items are selected according to a user defined priority, (ii) the on-line search, insert and delete operations are time optimal, and (iii) the data structure itself is space optimal. The solution presented can be used to reduce redundant

computations in large-scale scientific applications and, in addition, to simplify other non-numerical applications.

Despite its many limitations, FORTRAN remains the most commonly used language in scientific applications, especially for numerically intensive tasks. This is so because FORTRAN compilers are well-developed and therefore produce codes that execute very efficiently. In addition, the use of FORTRAN ensures compatibility with the largest number of existing scientific subroutine libraries. This compatibility requirement places two restrictions on the data structure, namely, a fixed-size or static-array constraint since dynamic storage allocations are not allowed and a restriction to non-recursive program logic [Mac 83]. To appreciate the significance of these conditions recall that although an AVL [Ade 62, Aho 74, Fos 73, Kro 79, Wir76] tree is known to be a very useful dynamic data structure that supports efficient on-line update operations, its use in a FORTRAN environment is very limited because a non-recursive delete algorithm is not generally available. On the positive side of these issues, on the other hand, it is important to note that even if one runs in an environment that allows for dynamic storage allocation it may be desirable to cap the database at some large but finite size so possible run-away situations can be avoided. Furthermore, a non-recursive programming logic can effect significant gains in any environment as it reduces system overhead caused by recursive calls.

The need for and value of a priority directed delete operation when the application is constrained to a fixed-size database should not be overlooked. For large-scale scientific applications it is desirable to be able to add a new data item to a data structure that has already reached its space limitation,

specifically, one that is more costly to calculate than an item already included. By incorporating a user supplied function for assigning priorities to data items, those items which are most valuable can be retained. In particular, observe that the time required for a routine to execute may vary over several orders of magnitude, depending upon the input variables. So even though the final stored results may appear similar, for example, a fixed set of floating point numbers, their relative values as measured by generation time can be very different. However, this difference in generation time may be offset by use frequency. For example, if result A takes ten times longer to generate than B, but is used only once, while B is used fifty times, then B is clearly the preferred element to be retained. Scenarios of just this type dictate the need for an external "intelligent" user supplied scheme for determining priorities of database entries if recalculation costs are to be minimized.

This chapter is organized according to the six topics listed below. In addition, as part of the final section, the significance, implications, and possible generalizations of the research are discussed. The results and order in which they are presented follow:

(1) A new data structure called a weighted search tree (henceforth referred to by the acronym *wst)* is introduced.

(2) The least-frequently-used and lowest-base-priority concepts are integrated into a time-space efficient priority scheme for the *wst.*

(3) A non-recursive delete algorithm for AVL trees that is time optimal and maintains all tree properties is given.

(4) A new representation T_L for binary search trees which is more flexible and powerful than the conventional representation T is introduced.

(5) Proof that the *wst* can be implemented in optimal time and optimal space by combining (1), (2), (3) and (4) is given.

(6) Test results on the performance of the *wst* are presented which show that the new representation T_L [Zhe 89] not only reduces the space requirement but also improves its time efficiency.

2.2 Weighted Search Tree

As mentioned in Section 2.1, our numerical database model consists of at least the following attributes:

KEY:	an element from a predefined set with total ordering;
INFORMATION:	a data item associated with the KEY value, and
PRIORITY:	a number that specifies the weight (relative importance) of the KEY value and its associated INFORMATION item.

We assume that there is a one-to-one mapping from the KEY set to the INFORMATION set. The total ordering of KEY values allows one to uniquely identify the associated INFORMATION and PRIORITY values; however, the same INFORMATION and/or PRIORITY values may be associated with two different KEY values. The PRIORITY values are used to weight the importance of the KEY values and therefore indirectly the importance of the

corresponding INFORMATION values. In numerical applications, we assume that the KEY is a variable and the INFORMATION values are results computed for different variable values. Let K be a given KEY value and I(K) and P(K) be the INFORMATION and PRIORITY values associated with K, respectively. The P(K) values are defined as numbers whose magnitudes reflect the time required for computing the I(K) and/or the use frequency of K.

The operations supported by the numerical database are as follows:

(1) Search(K): find the key value K in the database. If K is found, then retrieve I(K) and update P(K).
(2) Insert(K): compute I(K) and the initial P(K). Add K, I(K) and P(K) to the database.
(3) Deletemin: delete entry K with minimum P(K) from the database.

Clearly, all these are on-line operations. We insist that an insertion can only be invoked upon an unsuccessful search, and deletemin can only be invoked when the database storage limit is reached and an insertion is necessary. The lower bound on time for each on-line search operation on a set of n elements is $\Omega(\log n)$ [Knu 73a, Knu 73b]. Likewise, a height balance search tree can be used to ensure that insertions are carried out in $O(\log n)$ time. It is also known that to maintain a height balanced search tree with random on-line insertions and deletions requires $\Theta(\log n)$ time [Kro 79, Aho 74]. Therefore, the optimal time for operations (1), (2) and (3) is $\Theta(\log n)$. To achieve this time optimality, it is natural to represent a numerical database by two back-to-back height balanced search trees, one for the key values, and the

other for priority values. Since each node in a tree needs storage for pointers, we chose to use a priority queue (heap) for PRIORITY values to reduce the storage requirement without losing the time efficiency.

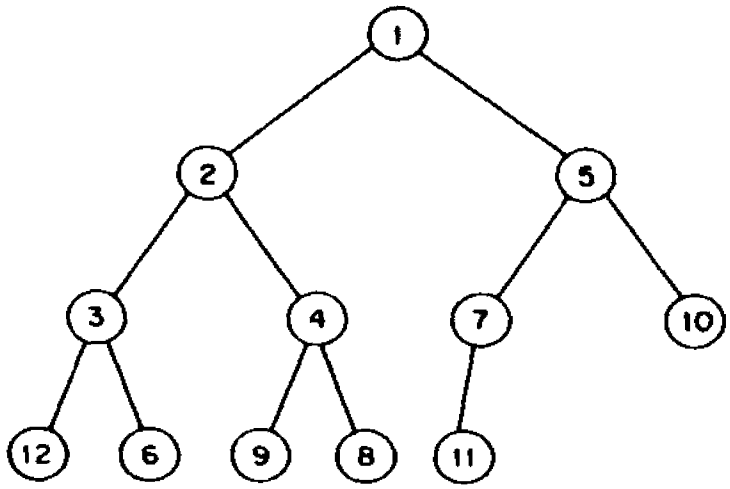
A heap is an implicit complete binary tree represented by a linear array [Lip 86, Hor 78]. Storage for pointers in a heap is not required as the location of left child, right child and parent elements can be deduced from the order of its elements. For example, if the location in a heap of the i-th element is specified by $l(i)$, then the location of the corresponding left child, right child and parent elements are $2 \times l(i)$, $2 \times l(i) + 1$ and $\lfloor l(i)/2 \rfloor$, respectively. For an implicit binary tree heap (minheap), the priority value of a node is not larger than the priority values of all of its decedent nodes. Suppose that there are n elements in a heap. When a new element p is inserted, it will be put in the (n+1)-th position in the linear array. Then, by a sequence of comparing and interchanging operations with p's current parent element, the element p will finally reach a position in the array such that the partial ordering heap property is satisfied. The deletemin operation is as follows: by the heap property, the first element in the array is deleted and then by putting the current last element p in the first position of the array and applying a sequence of comparing and interchanging operations with p's current children elements, p will finally reach a position in the array such that the partial ordering heap property is satisfied. The process of a sequence of comparing and interchanging operations is called heap adjustment. Clearly, when a new element is added, the heap adjustment takes $O(\log n)$ time. Similarly, in $O(1)$ time the minimum element can be found and deleted, and the heap property can be reinforced in $O(\log n)$ time [Lip 86] by the heap adjustment procedure. Therefore, a heap is an ideal choice to ensure

the time and space optimalities of a numerical database. In Figure 2.1a, we give an example of the binary tree representation of a heap consisting of 12 elements, with a priority value. Figure 2.1b illustrates how the heap in Figure 2.1a is represented by a linear array.

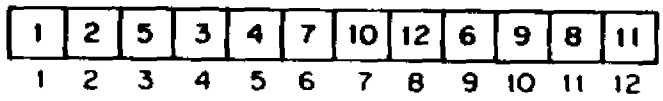
Based on this information we know that the combined data structure of a heap and a height balanced search tree guarantees time optimal on-line search, insert and delete operations for our numerical database. We call this structure a *weighted search tree* (*wst*). There are several variations of height balanced search trees, among them are AVL trees, weight balanced trees and 2-3 trees [Knu 73a, Knu 73b]. Whereas AVL trees and weight balanced trees are binary trees, 2-3 trees are ternary trees. Since binary trees can be represented more efficiently than ternary trees in FORTRAN, 2-3 trees appear not to be a good choice for our weighed search tree. Similarly, for space efficiency, AVL trees are superior to weight balanced trees, since each AVL tree node needs two bits to represent the balancing information (balance factor with value 0, -1 or +1) and each node in a weight balanced tree needs a full word to represent the balancing information (weight). Therefore we choose an AVL tree for our *wst*.

Consider a *wst* organized as six linear arrays: KEY[1..max], INFORMATION[1..max], LCHILD[1..max], RCHILD[1..max], BF[1..max] and PRIORITY[1..max], where in each of these *max* is the predetermined (constant) size of the database. The five *wst* arrays identified as KEY[1..max], INFORMATION[1..max], LCHILD[1..max], RCHILD[1..max] and BF[1..max] represent an AVL tree ordered on the key values KEY[i]. In

Figure 2.1 Heap. a) Binary tree structure of a heap. A minheap is used, that is, the value of a parent node in the heap is less than those of its left child and right child. b) Linear array representation of a heap. When $1 \leq i \leq \lfloor L/2 \rfloor$ where L is the length of the linear array, the left child and the right child of i-th element in linear array are the $2 \times$ i-th and $2 \times i + 1$-th elements, respectively.



(a)



(b)

addition to the value of the key, KEY[i], node i also has associated with it a data item, INFORMATION[i], computed from variables used to determine KEY[i]. The array entries LCHILD[i] and RCHILD[i] are indices that point to the left child and the right child of the i-th node and BF[i] is its balance factor. We use a variable named ROOT for the index of the root node of the tree. The priority value of i-th node is PRIORITY[i]. Search operations on the *wst* can be carried out easily by binary searches starting from the root node. However, insert and deletemin operations are not as straightforward. Inserting a new node involves two tree adjustment procedures, one for enforcing the partial order property of the heap, and the other for AVL tree rebalancing. It is important to note that during the heap adjustment process, when a node k interchanges its position with the new node, the LCHILD or RCHILD fields of the parent node of node k need to change to maintain the AVL tree structure. Finding the parent of k requires searching from the root of the AVL tree, which takes $\Omega(\log n)$ time [Kro 79, Aho 74], where n is the number of nodes in the tree. Since it is possible that $O(\log n)$ time is needed to find a parent node in the process of heap adjustment, a worst case scenario is that $O(\log^2 n)$ time is required for the heap adjustment process. To ensure $O(1)$ time for finding the parent node of an arbitrary node k, an additional field PARENT[k], which contains the index of the parent node of k, needs to be associated with the k-th node. Clearly, with this PARENT field the total time required for an insertion is $O(\log n)$. Similarly, the deletemin operation can be decomposed into two suboperations: delete the minimum priority node from PRIORITY[1..max] and the AVL tree, which involves an AVL tree rebalancing process, and then apply

the heap adjustment procedure. The additional PARENT field ensures that a deletemin operation can also be carried out in $O(\log n)$ time.

In Figure 2.2, we show an AVL tree of twelve nodes. Each arc labeled L (R) emanating from a node is a pointer pointing to the left (right) child of that node. The ground symbol indicates a null pointer. Each node is labeled by an integer pair (x,y), where x is the KEY value and y is the PRIORITY value. The traversal sequence of the KEY values is linearly ordered. By definition of an AVL tree, this is a binary search with respect to x. In Figure 2.2, we also include the balance factors (-1, 0 and +1) associated with the tree nodes. In Figure 2.3, we show a *wst* represented by the linear arrays KEY, PRIORITY, LCHILD, RCHILD and PARENT. This representation corresponds to the binary search tree in Figure 2.2. Since the INFORMATION field is not relevant to our discussions, it is not included. We also ignore the BF field for the moment. We will show how to implement it in Section 2.4. The array indices are the addresses of the tree nodes. The KEY value, PRIORITY value, the addresses of the left child, the right child and the parent node of node i are KEY[i], PRIORITY[i], LCHILD[i], RCHILD[i] and PARENT[i], respectively. Zero values in the LCHILD, RCHILD and PARENT fields represent null pointers. The array PRIORITY is a heap identical to Figure 2.1. The AVL tree for the KEY values is the one given in Figure 2.2, in which the (x,y) pairs specify the relationship between KEY values and PRIORITY values.

A *wst* supports optimal operations specified for our numerical database model. We use the implicit tree structure of a heap to save space; however, as discussed above, to enforce the time optimality, additional space for PARENT[1..max] is necessary. In Section 2.5, we introduce a new

Figure 2.2 An AVL tree for x of pairs(x, y). Each arc labeled L (resp. R) from a node is a pointer pointing at the left (resp. right) child of the node. The ground symbol indicates a null pointer. Each node is labeled by an integer pair (x, y), where x is the KEY value and y is the PRIORITY value. The number above the circles on the tree are the balance factors, -1 if the subtree to the right is one node longer than the one to the left, 0 if the left and right subtrees are the same length, and +1 if the subtree to the left is one node longer than the one to the right.

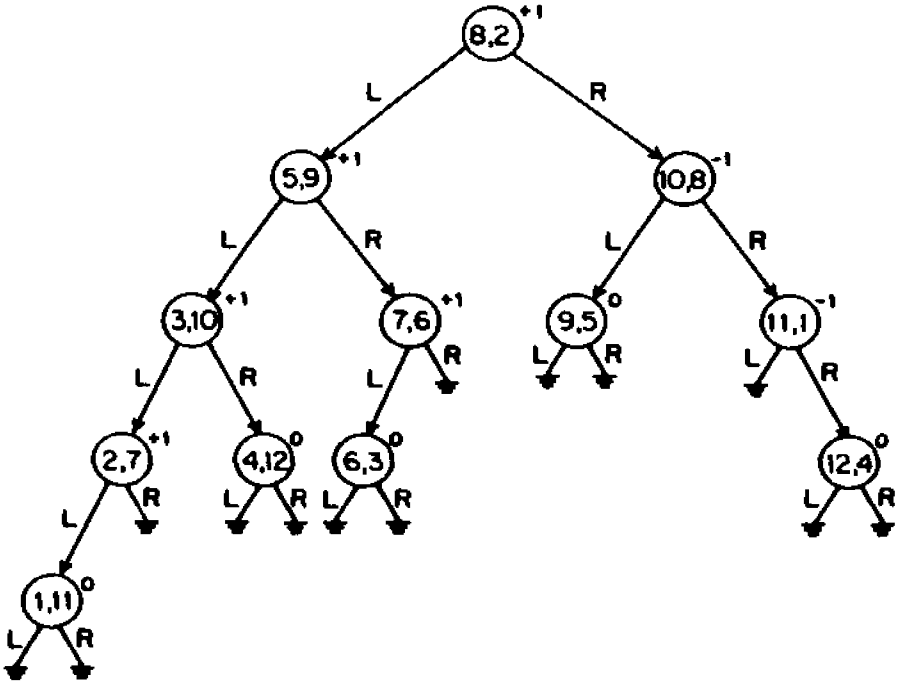


Figure 2.3 A weighted search tree. A *wst* represented by linear arrays KEY, PRIORITY, LCHILD, RCHILD and PARENT is shown. This representation corresponds to the binary search tree in Figure 2.2. The order of array shows a priority queue (heap) same as in Figure 2.1.

	1	2	3	4	5	6	7	8	9	10	11	12
PRIORITY	1	2	5	3	4	7	10	12	6	9	8	11
KEY	11	8	9	6	12	2	3	4	7	5	10	1
LCHILD	0	10	0	0	0	12	6	0	4	7	3	0
RCHILD	5	11	0	0	0	0	8	0	0	9	1	0
PARENT	11	0	11	9	1	7	10	7	10	2	2	6

representation of binary search trees to show how to eliminate the space required for PARENT[1..max].

2.3 Priority Strategy

The effectiveness of a numerical database system implemented with a *wst* not only depends on the efficiency of search, insert and delete operations, which is the subject to be discussed in the next section, but also depends on how the priority values are defined and updated. A good priority strategy should enable the frequency of use of a data item and its intrinsic value to be incorporated into its assigned priority. We propose a scheme that integrates the least-frequently-used [Pet 85] and lowest-base-priority concepts into the algorithm for assigning priority to the data items.

We define a priority scheme consisting of two parts: *base priority*, which reflects the computation time involved in generating the INFORMATION value associated with KEY, and the *hit frequency*, which measures the frequency of use of the KEY (and INFORMATION) values. Specifically, when a KEY value is accessed, its PRIORITY value is updated by adding its base priority to its current PRIORITY value to get its new PRIORITY value. And when a deletion is required, the node with the lowest PRIORITY value in the database is the one that is selected. To reduce storage requirements, the base priority and the hit frequency for each data item in the database are incorporated into its assigned priority. This is achieved by exploiting the bit representation of integer variables and the advantages that can be gained through the use of the bit-shift operations. Specifically, the

PRIORITY value assigned to each KEY value in the database is stored as a 32 bit integer variable with the first 8 bits reserved for the "base priority" and the remaining 24 bits for the "hit frequency × base priority" value. The actual value assigned for the PRIORITY is therefore the "hit frequency × base priority × 2^8 + base priority". For example, suppose a particular data item in the database has a base priority of 7 and a current hit frequency of 16, then the current assigned PRIORITY value is simply 16 × 7 × 256 + 7 = 28679, which has the following 32 bit representation: (00000000 00000000 01110000 00000111).

Although two multiplications and one addition are implied in this expression for determining the priority, the multiplications can be replaced by more efficient bit-shift operations. Specifically, whenever an element is hit, a bit-shift system call (ISHFT for IBM 3090/600E and DEC VAX-11/750 systems) [IBM 87] is first used to extract the base priority from the right-most 8 bits of the current PRIORITY value. Then another bit-shift operation plus an add suffices to give the updated PRIORITY value: NEW = OLD + ISHFT(ISHFT(OLD,24),-16). This expression first shifts the current PRIORITY value, the integer variable labelled OLD in the above result, left 24 bits to isolate the base priority, and then shifts it 16 bits to the right so upon addition to OLD the NEW PRIORITY value is obtained.

In this scheme base priorities are bound to the range 0 to 255 = 2^8-1. Of course, this can be extended but doing so decreases the range of the PRIORITY values accordingly. Note that with this scheme in place the priority saturates at its maximum value after $2^{(32-8-8)} = 2^{16}$ hits. Since 2^{16} = 65,536 is a very large number, this saturation effect is normally of little consequence.

Nonetheless, in the current application of the theory a saturated PRIORITY value retains its maximum value upon subsequent hits. In the general case, if N bits are assigned to the base priority, (32 - N) bits remain for the cumulative PRIORITY value and saturation sets in after $2^{(32-2N)}$ hits. If all nodes carry the same base priority, N can be set to one and the scheme simplifies to one in which the priority is determined by the hit frequency only.

The base priority for data items in a *wst* can be defined either internally, when the element is generated, or externally by the user. Internally defined base priorities would normally use some measurable quantity or quantities associated with the cost of producing and maintaining the date item, such as CPU time involved in the calculation and/or storage required for holding the INFORMATION value associated with the KEY. External base priorities, on the other hand, might be set by the user according to the type of computation. The priority strategy introduced here favors the data item with the largest base priority. For example, suppose that KEY A has a base priority of 10 and has been hit 20 times, while KEY B has a base priority of 20 and has been hit 10 times. The bit representations of these two will be identical beyond the right-most 8 bits, but the priority of node B will be greater than that of A because the former has a larger value in the right-most 8 bits. Of course, if KEY A is hit 21 times, then its total priority will be greater than that for KEY B.

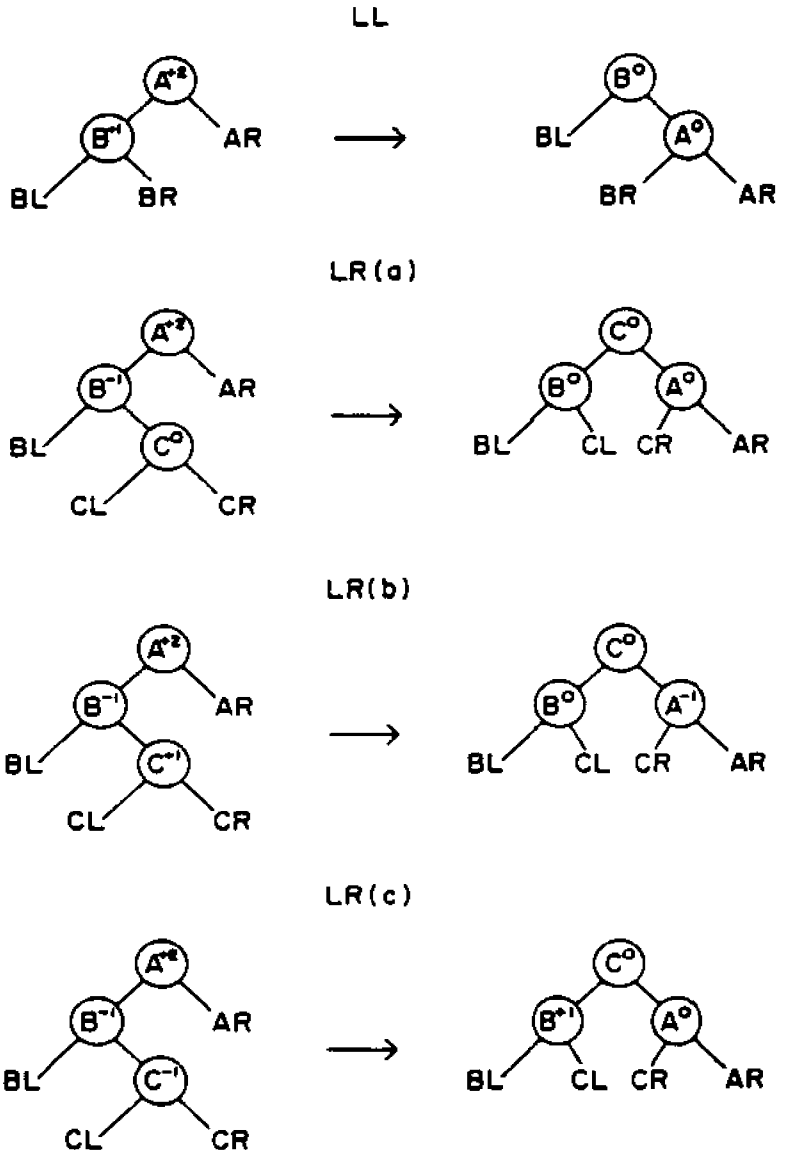
2.4 Search, Insert and Delete Algorithms

The search/insert algorithm carries out the dual purpose of extracting information from an existing node or, if the data item being sought is not found,

adding a new node to the *wst*. Whenever an item is inserted, the algorithm checks to ensure the tree is balanced. If it is not, the tree is rebalanced using one of several possible rotations [Hor 83]. The node insert operation on the AVL tree requires at most one rotation to maintain a balanced binary tree [Hor 83]. There are four different rotation types: LL, RR, LR and RL, all of which are symmetric. Figure 2.4 shows the LL and LR rotations. For the strategy described in the previous section, the PRIORITY value of an item on insertion is simply its own base priority, but this is increased with every subsequent fetch. Once the priority value is updated, the PRIORITY array is rearranged to be a valid heap.

Similarly, a delete operation on a *wst* requires a deletion from the AVL tree followed by a corresponding deletion from the heap. A recursive algorithm for the deletion of a node from an AVL tree followed by bottom-up rebalancing is well known [Wir 76]. The bottom-up approach requires that, after the deletion of a node, the tree is rebalanced from a leaf of the severed subtree to a proper node whose balance factor is zero. The algorithm introduced here is an iterative procedure that employs top-down rebalancing. This is achieved by identifying a node (denoted as x below) that lies on the path and is closest to the deleted node with either (a) a balance factor of 0, (b) a balance factor of +1 with a left child having a balance factor of 0, or (c) a balance factor of -1 with a right child having a balance factor of 0, and then rebalancing the tree from that node down to a proper leaf. After the lowest priority node (root of a heap) is deleted by the AVL tree delete algorithm, the linear array for maintaining priority must be rearranged to be a valid heap.

Figure 2.4 AVL tree rotation. The LL and LR types are shown for AVL tree rotations. RR and RL are symmetric to LL and LR, respectively.



While the insertion of a node into a tree requires at most one rotation, if the node being deleted is in the middle of a tree it can require several rotations. For the recursive algorithm described in [Wir 76] there are four types of rotations required for an insert operation. As indicated above, these four types are labelled LL, RR, LR and RL. The first two of these, LL and LR rotations, are shown in Figure 2.4. All four of these are needed for a *wst* plus an additional LL rotation that is shown in Figure 2.5 and its symmetric RR partner that is not shown. We give the delete algorithm as follows, in which node y is to be deleted.

Delete Algorithm

Step 1. Find x along path from the root to node y:

```
p := ROOT;
while (p ≠ y) do
    if (p satisfies one of the conditions for x) then x := p;
    if (KEY[p] > KEY[y]) then p := LCHILD[p]
    else if (KEY[p] < KEY[y]) then p := RCHILD[p];
endwhile
```

Step 2. Delete node y in the linear tree array:

```
if (LCHILD[y] = null) then
    if (RCHILD[y] ≠ null) then y := RCHILD[y]
    else y := null
else if (RCHILD[y] = null) then y := LCHILD[y]
else begin
```

z := the node contains the largest key value smaller than $KEY[y]$;

$i := y$;

while ($i \neq null$) **do**

if (i satisfies one of the conditions for x) **then** $x := i$;

if ($KEY[i] > KEY[z]$) **then** $i := LCHILD[i]$

else if ($KEY[i] < KEY[z]$) **then** $i := RCHILD[i]$;

endwhile

$y := z$;

end

Step 3. Balance the tree:

while (x is not a leaf) **do**

if (height difference of subtrees of $x > 1$) **then** *rotate*;

if ($KEY[y] < KEY[x]$) **then** $x := RCHILD[x]$

else x := $LCHILD[x]$;

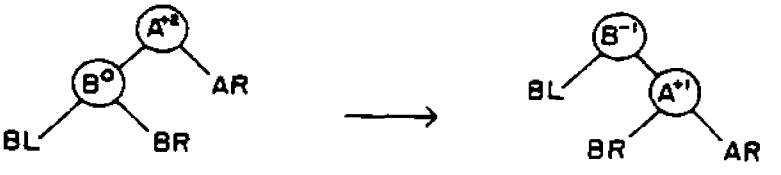
endwhile

It is easy to verify the correctness of this algorithm and that its complexity is $O(\log n)$. For simplicity, we omit the proof.

2.5 New Representation of Binary Search Trees

As mentioned in Section 2.2, for efficient dynamic maintenance of a *wst*, each node i needs an additional field PARENT[i]. In this section, we introduce a simple and flexible multilist representation T_L [Zhe 89] for binary search trees and show that by using this representation finding the parent of any given

Figure 2.5 NEW LL rotation. When the delete operation is performed a new LL rotation is sometimes necessary. The complementary new RR rotation is symmetric to new LL rotation and therefore not shown.



NEW LL ROTATION

tree node requires $O(1)$ time. In fact, our conclusion is much stronger than this. We show that any algorithm on conventional binary search trees with time and space complexities $O(t(n))$ and $O(s(n))$ can be converted into an algorithm performing the same operations with time and space complexities $O(t'(n))$ and $O(s'(n))$ on their corresponding representations T_L such that $O(t'(n)) \leq O(t(n))$ and $O(s'(n)) \leq O(s(n))$ with the inequality holding for some operations.

As with conventional binary search trees, each node i in our representation T_L of a binary search tree consists of three fields: KEY[i], LLINK[i] and RLINK[i], where LLINK and RLINK are used as the LCHILD and RCHILD fields in a conventional binary search tree, but used in a different way in the new *wst* representation. Let T be the conventional representation of a binary search tree. We assume that for a node i in T, LLINK[i] = null (RLINK[i] = null) if and only if i has no left (right) child, where null is a special value that can not be used as a normal pointer. Referring to T, our representation T_L is defined as follows:

(1) if i is the root of T, then RLINK[i] = null in T_L;

(2) if i does not have a child in T, then LLINK[i] = null in T_L;

(3) if j is the left child of i in T, then LLINK[i] = j in T_L; or if i does not have a left child but j is the right child of i in T, then LLINK[i] = j in T_L;

(4) if j is the left child of i and i does not have a right child in T, then RLINK[j] = i in T_L; or if j is the right child of i in T, then RLINK[j] = i in T_L.

Graphically, if we consider every node of T_L as a vertex and every non-null link of nodes in T_L as an arc, then T_L is a graph with directed cycles. Note that T_L uses exactly the same amount of space as its corresponding T. In Figure 2.6 we give our representation T_L for the binary search tree T shown in Figure 2.2. Note that these tree representations are with respect to the values x in all pairs (x, y) used for labelling the nodes. Assuming that all key values in T_L are distinct, it is easy to verify the following properties of T_L, as compared with its corresponding conventional binary search tree T:

(i) i is the root of T if and only if RLINK[i] = null in T_L;

(ii) i is a leaf in T if and only if LLINK[i] = null in T_L;

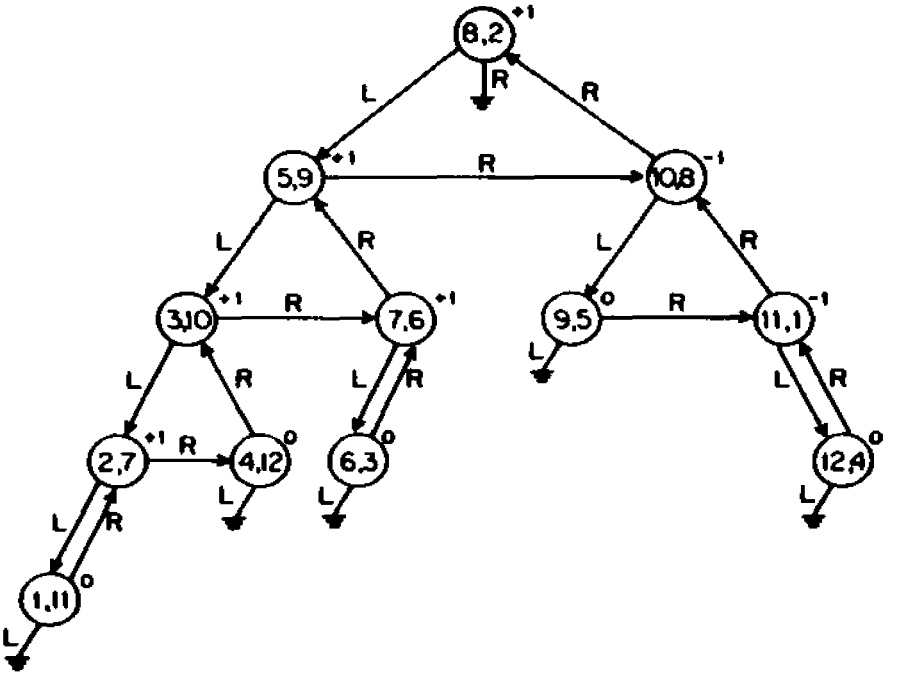
(iii) i has exactly one child in T if and only if LLINK[i] ≠ null and RLINK[LLINK[i]] = i in T_L, and this child is LLINK[i] in T_L; furthermore, if i has a left child in T, then KEY[LLINK[i]] < KEY[i] in T_L, otherwise i has a right child in T;

(iv) i has a left child in T if and only if LLINK[i] ≠ null and KEY[LLINK[i]] < KEY[i] in T_L, and furthermore, this node in T_L is LLINK[i];

(v) i has both the left child and the right child in T if and only if LLINK[i] ≠ null and KEY[LLINK[i]] < KEY[i] < KEY[RLINK[LLINK[i]]] in T_L, and furthermore, these child nodes are LLINK[i] and RLINK[LLINK[i]], respectively, in T_L;

(vi) j is the parent node of i and i is the right child of j in T if and only if j = RLINK[i] ≠ null and KEY[i] > KEY[j] in T_L;

Figure 2.6 T_L representation of an AVL tree for x of pairs (x, y). A multilist representation of the AVL tree in Figure 2.2 is given. The notation is the same as in Figure 2.2 except that here L's and R's represent the left link and the right link, respectively. Using the left link and the right link, the positions of the left child, the right child and the parent node can be computed.



(vii) j is the parent node of i and i is the left child of j in T if and only if one of the following conditions holds:

(a) j = RLINK[i], KEY[i] < KEY[j], and RLINK[j] = null;

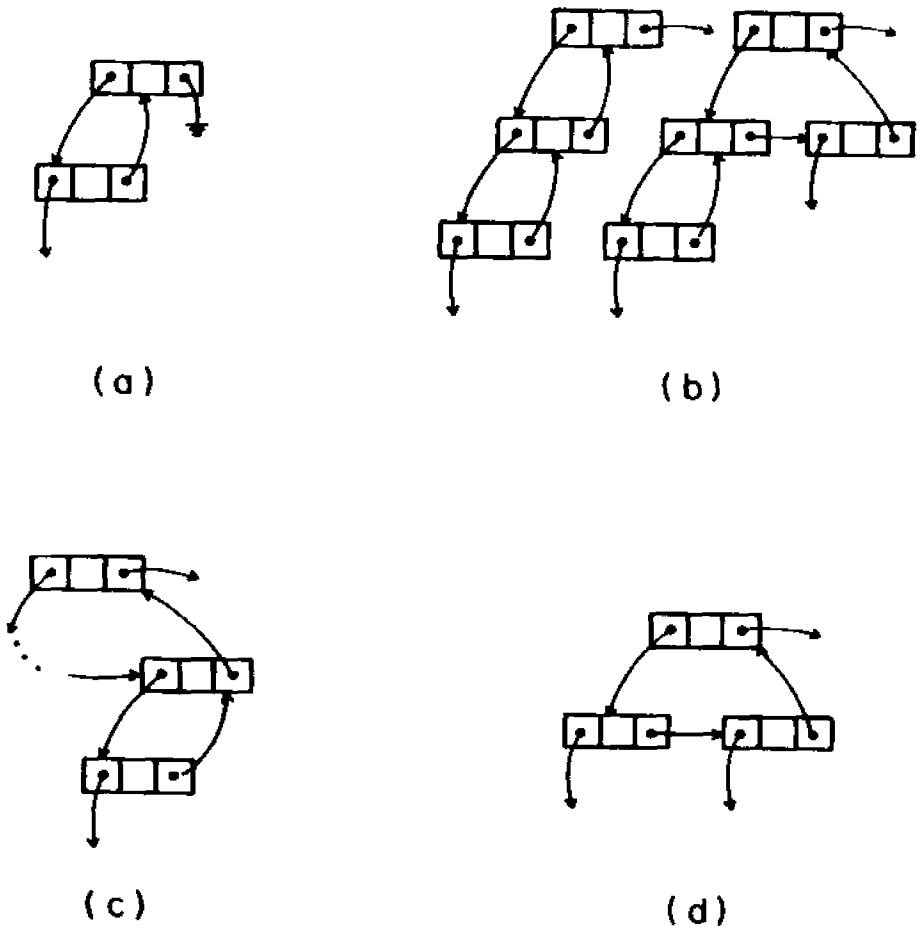
(b) j = RLINK[i] and KEY[i] < KEY[j] < KEY[RLINK[j]];

(c) j = RLINK[i], KEY[RLINK[j]] < KEY[i] < KEY[j]; and

(d) j = RLINK[RLINK[i]], KEY[i] < KEY[RLINK[i]] and KEY[j] < KEY[RLINK[i]].

All of these properties of T_L can be easily proved by using the linear order of the keys associated with the nodes of T. For example, for property (vii), the situations corresponding to (a), (b), (c) and (d) are shown in (a), (b), (c) and (d) of Figure 2.7, respectively. For simplicity, we omit the proof for all the above listed properties. These properties can be divided into three groups. The first group, group A, which consists of (i) and (ii), can be used to determine whether or not a given tree node is the root or a leaf. The second group, group B, which consists of (iii), (iv) and (v), can be used to find the left and/or the right child of a given node. The third group, group C, which consists of the remaining properties, can be used to find the parent node of a given node and determine if the given node is a left or right child of its parent. By these properties, one can show that T_L is more powerful than its corresponding T. Now let us make the following comparisons between T_L and its corresponding T:

Figure 2.7 Conditions for finding a parent node. These four conditions show how to find a parent node in the T_L representation.



(1) As mentioned earlier, for a given node i in T finding its parent node needs $\Omega(m)$ time, where m is the length of the path from the root of T to i. For the same operation on T_L, $O(1)$ time is sufficient by the properties in group C. Note that this efficiency is achieved without sacrificing space.

(2) Other operations on T_L can be implemented by mimicking their counterparts on T. For operations such as search, insert and delete, it is easy to see that the properties in groups A and B are sufficient. Thus, these operations on T_L have the same time complexities as their counterparts on T. Similarly, any tree balancing algorithm for conventional binary search trees can be converted into an algorithm for T_L with the same complexity. Therefore, $O(\log n)$ time for search, insert and delete operations (including tree rebalancing after updating T_L) is attainable for T_L.

(3) All the above listed properties of T_L allow us to easily convert a recursive operation on T into a non-recursive one on T_L without using stacks. Consequently, not only is the space for an implicit or an explicit stack saved, the system overhead for executing a recursive algorithm is also reduced.

(4) Using all the above listed properties of T_L, one can develop $O(n)$ time inorder, preorder and postorder tree traversal algorithms using $O(1)$ additional space in straightforward ways. In fact, these three traversal algorithms are trivial using recursion. By (3), these recursive operations can be transformed into iterative ones. To our knowledge, no algorithms for preorder and postorder traversing conventional binary

search trees in $O(n)$ time and $O(1)$ additional space without altering and recovering the original tree links during the tree traversal have been reported.

Using this new representation, the PARENT field of a *wst* can be eliminated, as shown in Figure 2.8 for the *wst* of Figure 2.6. To find the parent of a node i in T_L, assuming that i is not the root of the tree, the following statement is sufficient:

```
if LLINK[RLINK[RLINK[i]]] = i then parent := RLINK[RLINK[i]]
else  parent := RLINK[i]
```

Since RLINK[i] needs to be computed only once, finding the parent of any node i in T_L is very efficient. By (3), we know that we can convert the existing bottom-up recursive delete-and-rebalance algorithm for AVL trees into a non-recursive one, instead of using the algorithm given in the previous section.

2.6 Experiments and Performance Characteristics

We implemented the proposed *wst* in two ways: first, Implementation I uses the data structure in a way similar to the example [Par 90b] given in Figures 2.2 and 2.3, that is, a PARENT field containing the array index of the parent node is employed; second, Implementation II uses a data structure similar to the example given in Figures 2.6 and 2.8, where the binary search

Figure 2.8 A weighted search tree using the T_L representation. This linear array representation of a *wst* has the same properties as the one shown in Figure 2.3. However, this linear array does not include the parent positions since parent positions can be computed using LLINK and RLINK.

	1	2	3	4	5	6	7	8	9	10	11	12
PRIORITY	1	2	5	3	4	7	10	12	6	9	8	11
KEY	11	8	9	6	12	2	3	4	7	5	10	1
LLINK	5	10	0	0	0	12	6	0	4	7	3	0
RLINK	11	0	1	9	1	8	9	7	10	11	2	6

tree representation T_L is used to eliminate the PARENT field. To make Implementations I and II comparable, the following conditions were enforced:

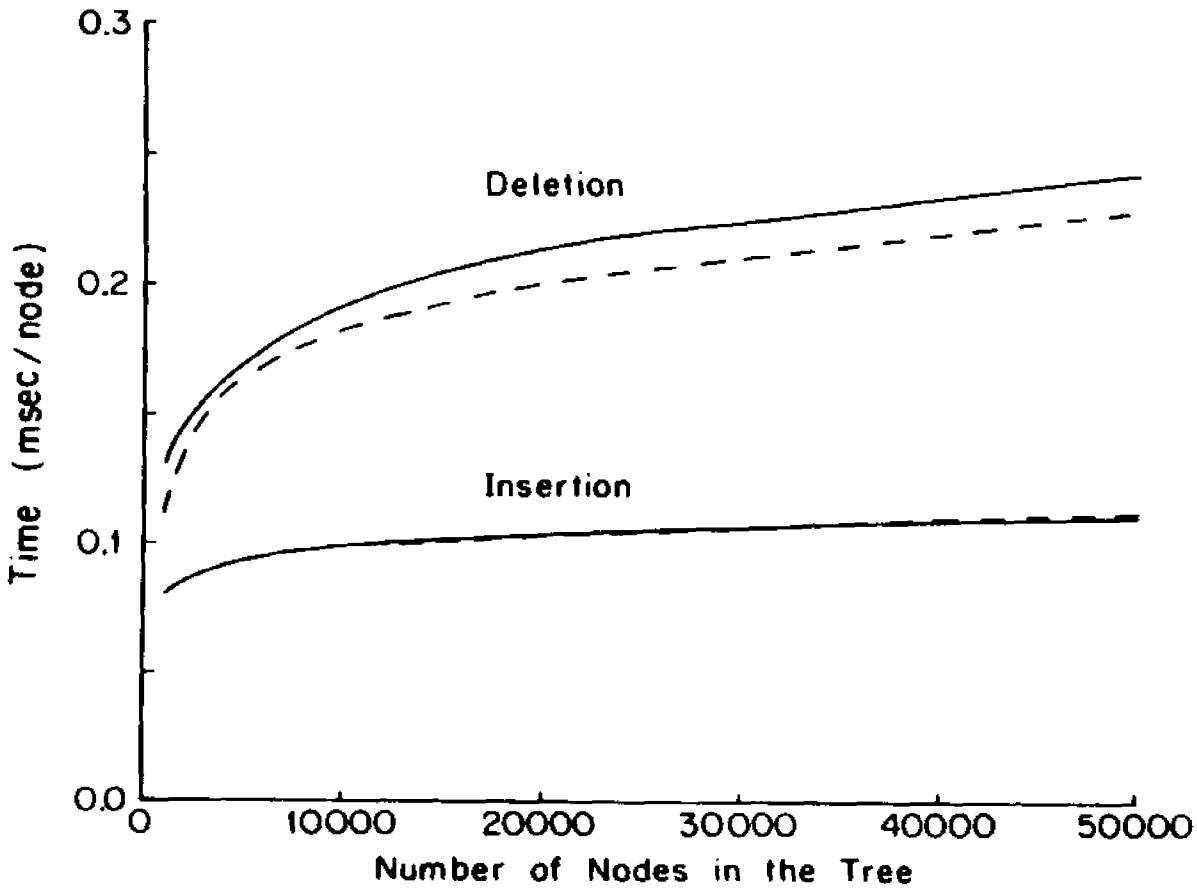
(1) The programs for these two implementations are identical, except for the ways of finding the left child, the right child and the parent node and updating the "pointer" fields;
(2) No information other than the KEY and PRIORITY values are stored in the nodes, since this is sufficient for the simulation;
(3) The priority function used in these two implementations is the same one described in Section 2.3;
(4) The sequence of operations, and the key and priority values involved in each operation, although generated randomly, are exactly the same for the two implementations.

As mentioned in Section 2.1, in general, it is possible to have multiple keys. In our implementations, we assume that there are five keys, each an integer generated by a simple random number generator, so that a large range of keys is possible. We have conducted search/insert and delete experiments using both implementations. Let *max* be the predetermined maximum number of nodes in a *wst*. The search/insert simulation is executed as follows: generate a quintuple of integer numbers as the key using a random number generator and perform a binary search of the tree; if the key is found in the tree then update its priority, otherwise insert the key into the tree and initialize its base priority value by calling for a number from the random number generator. This process continues until the limit *max* of nodes in the tree is reached. The delete

simulation is as follows: given a tree with *max* number of nodes generated by the search/insert procedure, repeatedly perform deletemin operations until the tree is empty. The average performance for search/insert and delete operations on an IBM 3090/600E is shown in Figure 2.9, where the solid curves corresponds to the simulation results of Implementation I and the dashed curves correspond to the simulation results of Implementation II.

It should be pointed out that the system overhead for the virtual memory management is taken into account as part of the performance. Nevertheless, since the difference in the memory requirements for these two implementations is insignificant, the effect of page faults, for example, can be ignored in comparing the two implementations. Theoretically, as indicated in Section 2.5, our new representation T_L for binary search trees is not as time efficient as its corresponding conventional representation T. Nevertheless, the results show that the performance of search/insert operations on a *wst* using T_L and T are about the same while for delete operations using T_L produces about a 6~7% gain over T. An explanation of this phenomenon is that both heap adjustment and AVL tree rebalancing procedures for a *wst* need to directly or indirectly modify the linkage related to the parent nodes. However, the total number of comparisons and memory accesses for Implementation II is less than that for Implementation I, insofar as heap adjustment and AVL tree rebalancing procedures is concerned. Since each insert operation causes at most one rotation operation, the saving in the AVL tree rebalancing procedure cancels out the added overhead associated with the binary search in Implementation II. This explains why the search/insert performances of two implementations are about the same. In contrast, each delete operation may

Figure 2.9 Experimental results for the *wst*. The average performance for search/insert and delete operations on an IBM 3090/600E is shown, where the solid curves correspond to the simulation results using the binary tree representation T and the dashed curves correspond to the simulation results using the representation T_L for a *wst*.



require several AVL tree rotation operations for rebalancing. In such situations, a saving in memory access operations is realized in Implementation II as compared to Implementation I. Note that the balance factor field BF can be eliminated by utilizing one bit of each of the two pointer fields or two bits of the PRIORITY field. Our experiments indicate that Implementation II, which uses our new binary search tree representation T_L, outperforms Implementation I, which uses conventional binary search tree representation, in both time and space efficiencies.

2.7 Discussion

The objective of this research was to provide a practical solution to the ever-present and difficult problem of reducing redundant calculations in large-scale scientific computing applications. The solution offered is a new data structure called a weighted search tree *(wst)* which is specifically designed for large-scale scientific applications. It uses non-recursive logic and respects the static-array requirement for FORTRAN compatibility. This data structure provides for time-optimal on-line search as well as insert and delete operations on a fixed-size numerical database using the combination of hit frequency and a user defined base priority function for assigning overall priorities to entries. By using the weighted search tree, the most valuable data items, as measured by priority values that reflect their computation time and use frequency, can be kept in fixed-size databases. As a result, redundant calculations of intermediate results can be avoided without over commiting space resources.

The new simple representation T_L for binary search trees introduced in Section 2.5 is flexible and powerful. By the properties of T_L, any algorithm on conventional binary search trees T with time and space complexities $O(t(n))$ and $O(s(n))$ can be converted into algorithms performing the same functions with time and space complexities $O(t'(n))$ and $O(s'(n))$ on the corresponding representations T_L such that $O(t'(n)) \leq O(t(n))$ and $O(s'(n)) \leq O(s(n))$ with the inequality holding for some special operations. These time and space efficiencies are achieved without introducing additional information and therefore space to nodes in the tree. We assumed that all key values are distinct. This assumption holds for most applications of binary search trees. One generalization of this representation is to allow for duplicate key values. A possible approach to accommodating duplicate key values is to construct a two-level data structure. The first level is a T_L, which contains all distinct key values, including a representative from each set of identical key values. The second level consists of clusters of keys, each cluster corresponds to a set of identical key values. We would like to point out that our representation T_L for binary search trees can be generalized into a rich set of variations of binary search trees. For examples, 2-3 trees, B-trees, interval trees, segment trees, multidimensional binary search trees, range trees, etc. (see [Aho 74, Knu 73a, Knu 73b, Ben 75, Pre 85] for details). We believe that similar results on these trees can be obtained.

The space efficiency of a *wst* is derived from the concept of implicit data structures. First of all, the heap component of a *wst* is an implicit data structure itself, since the "pointers" pointing to the left child, the right child and the parent node are not present but computed. Second, the AVL tree is

represented as a multilist structure organized in such a way that the "pointers" pointing to the left child, the right child and the parent node are computed using the semantics of the key values. Third, the priority scheme codes the base priority and hit frequency into one single value, and the base priority can be decoded by bit manipulation for updating the priority after every access of the associated information. In addition, the balance factor, which needs only two bits of each node of the AVL tree is coded into the two pointers of the node. By exclusively using the concept of implicit data structure, a *wst* is represented in optimal space for supporting specified search, insert and delete operations.

As a by-product of this research effort, an iterative node-delete algorithm for AVL trees was developed. Unlike the previously known recursive AVL tree node-delete algorithm, which rebalances the tree in a bottom-up fashion, the node-delete algorithm introduced here uses non-recursive logic and rebalances the tree in the opposite or top-down direction. Indeed, this appears to be the only non-recursive AVL tree delete algorithm currently available in the literature.

The priority concept is central to the issue of practicality of a numerical database. This is because the value of information associated with the data entries, as measured, for example, by generating CPU time, may vary tremendously. We presented a priority scheme which determines priority values in terms of a user defined base priority, which captures the information generation time, and the hit frequency for the information, which captures the locality of memory reference that may be present in many scientific applications such as matrix manipulations. Since the base priority is defined by the user

when a data item is first generated, the priority strategy can be easily adapted to meet the special needs of a particular application without a change in the logic of the program used to generate the database. This makes our numerical database an invaluable user-friendly software package for large-scale numerically intensive scientific applications.

CHAPTER 3
WEIGHTED SEARCH TREE IMPLEMENTATION (CODES FOR A NUMERICAL DATABASE SYTEM)

In the previous chapter a weighted search tree (*wst*) was introduced and shown to be time-space optimal. In this chapter we present a numerical database system that is based on the *wst* and which functions like a file directory system. The database system, which can be used to reduce the all too frequent occurrence of redundant calculations in numerically intensive computations, can be used to fetch, insert and delete items from a dynamically generated list in optimal time. List items are ordered according to a priority queue with the entry priority of each element set by the user and subsequently updated to reflect hit frequency. New items can be added to a database as long as there is space to accommodate them and, when there is not, the lowest priority element can be removed to make room for an incoming element with higher priority. The system acts passively on a database and can therefore be applied to any number of databases with different structures in a single application.

3.1 Introduction

Scientific computing applications frequently involve redundant calculations. This occurs either because the number of values that need to be calculated is too large to store in memory or they occur in unknown combinations so pregeneration, which would allow the values to be ordered

efficiently so a simple binary look up could be used, is impracticable or impossible. An on-line numerical database system [Dat 86, Par 89a, Par 90c] can be used to circumvent this problem as it allows an existing database to be searched for a particular element and the information used when it is found, but if it is not found the element can be generated and then added to the list so it can be reused as necessary at a later stage in the calculation. Typically this *modus operandi* yields a two fold gain, namely, needed results are calculated only once and unneeded results are never generated. Although this is clearly the procedure of choice, it is usually a nontrivial objective to achieve. The numerical database routines in the WSTREE package allow this to be achieved in a FORTRAN software environment.

The numerical database system introduced in this chapter has 3 operations: search, insert and delete. An abstract data structure called a *weighted search tree* (*wst*) [Par 90c], which is the combination of a priority queue (heap) [Lip 86] and a height balanced AVL tree [Aho 74, Hor 83, Kro 79, Par 89a, Par 90c], is used to maintain the system. Each element in a list is associated with a node in a binary tree which uses left link and right link pointers to specify, respectively, the left and the right subtrees of each node. By construction, the index of the left (right) subtree is less (greater) than the index of the parent node. This left-right structure applies to every node in the tree. By requiring the tree to be height balanced, optimal $O(\log n)$ time [Aho 74, Hor 83, Kro 79] for operations on lists of length n can be achieved. The height of any tree or subtree is defined to be the maximum number of steps from its root to its apex. If the height of the left subtree of a node is h_L and the height of the right subtree is h_R, then the tree is said to be height balanced

modulo one if and only if $| h_L - h_R | \leq 1$ for every node in the tree. The balanced factor (BF) of a node is defined to be $h_L - h_R$. For any node in an AVL-tree the balance factor is either -1, 0 or +1. The association of list elements with nodes in a tree is illustrated in Figures 3.1 and 3.2 which show, respectively, the insertion order for list elements together with its unbalanced and height balanced representations.

The data structure that is used for constructing our numerical database system is actually a modification of a height balanced binary tree. As indicated above, a normal height balanced binary tree has two links per node which point to the left child and the right child of the parent node. The new data structure, which is a multilist representation [Hor 83, Par 90c, Zhe 89] of a height balanced tree, also has two links per node. However, these two links allow one to determine not only the left child and the right child but the position of the parent node as well. This feature is important because maintaining the *wst* heap structure, which is required for superimposing on the tree an organization of the nodes according to their priority, requires parent node position information. A portion of a multilist representation of a tree is shown in Figure 3.3. The feature that two links in a multilist representation suffices to additionally compute the position of the parent node saves space and renders the system space optimal. Figure 3.4 is a multilist representation of the height balanced binary tree shown in Figure 3.2.

Our on-line database system has a delete operation. This is an essential feature when dealing with data structures that are either larger than the memory capacity of the machine being used or one is working in a programming environment that does not allow for dynamic allocation of storage.

Figure 3.1 Unbalanced binary tree. The original insertion order is indicated on the left. The tree on the right is an unbalanced binary tree constructed from the list on the left.

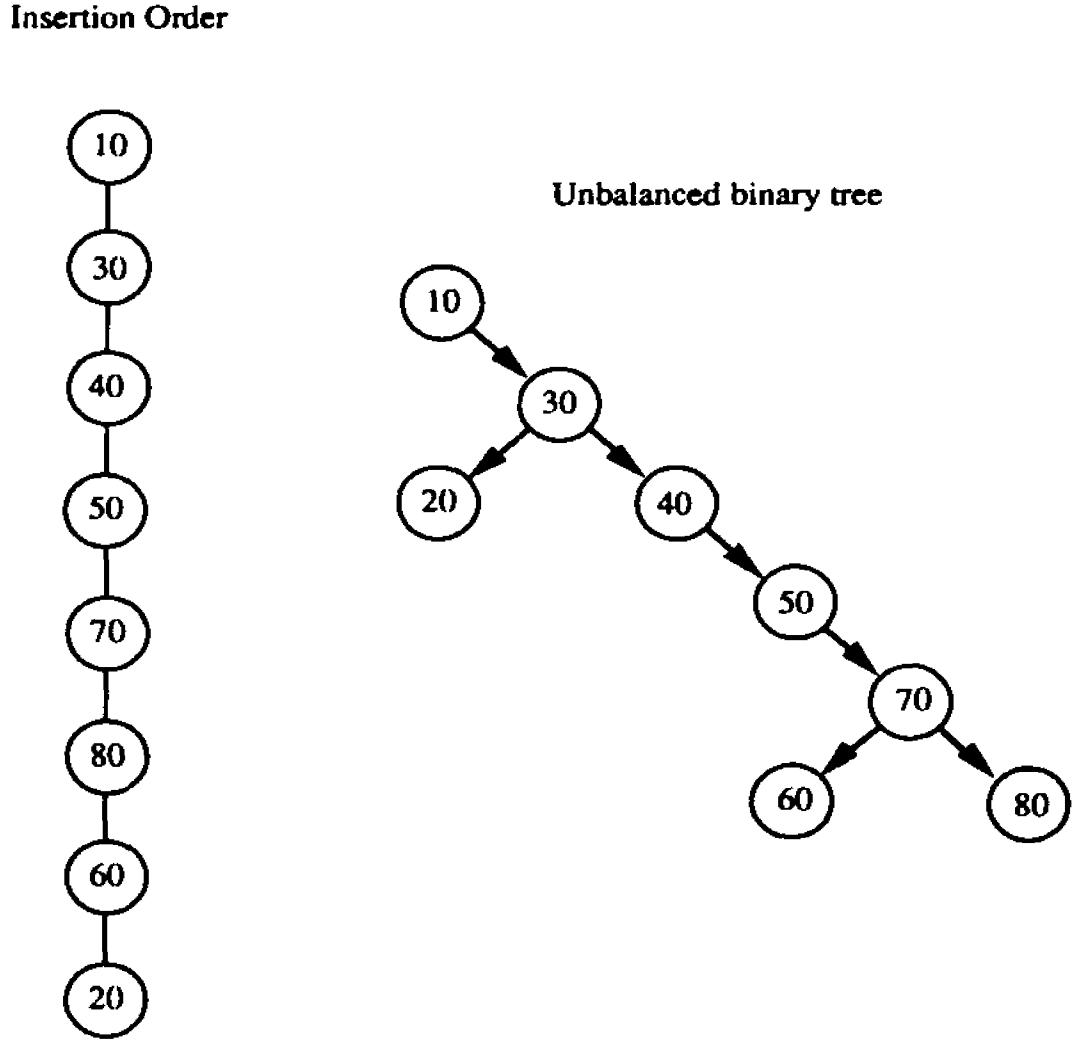


Figure 3.2 Height balanced binary tree. The number above the circles are the balance factors: -1 if the subtree to the right is one node longer than the one to the left, 0 if the left and right subtrees are the same length, and +1 if the subtree to the left is one node longer than the one to the right.

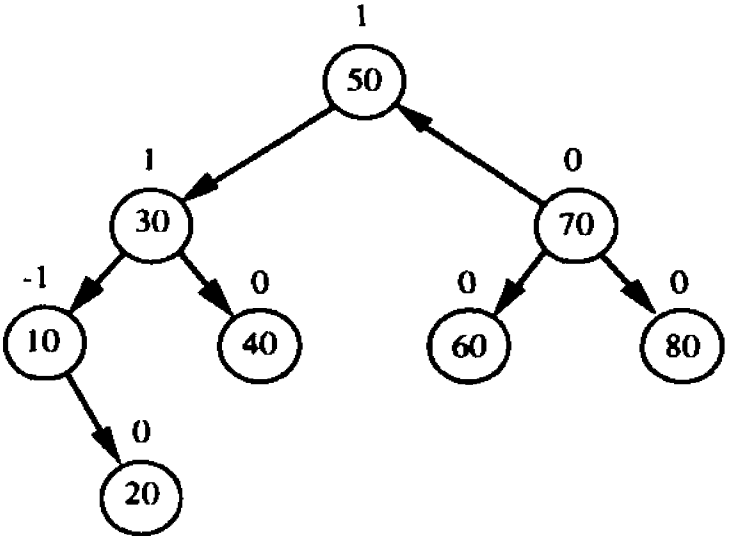
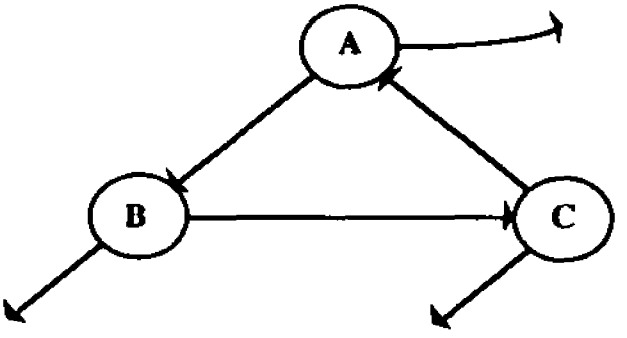
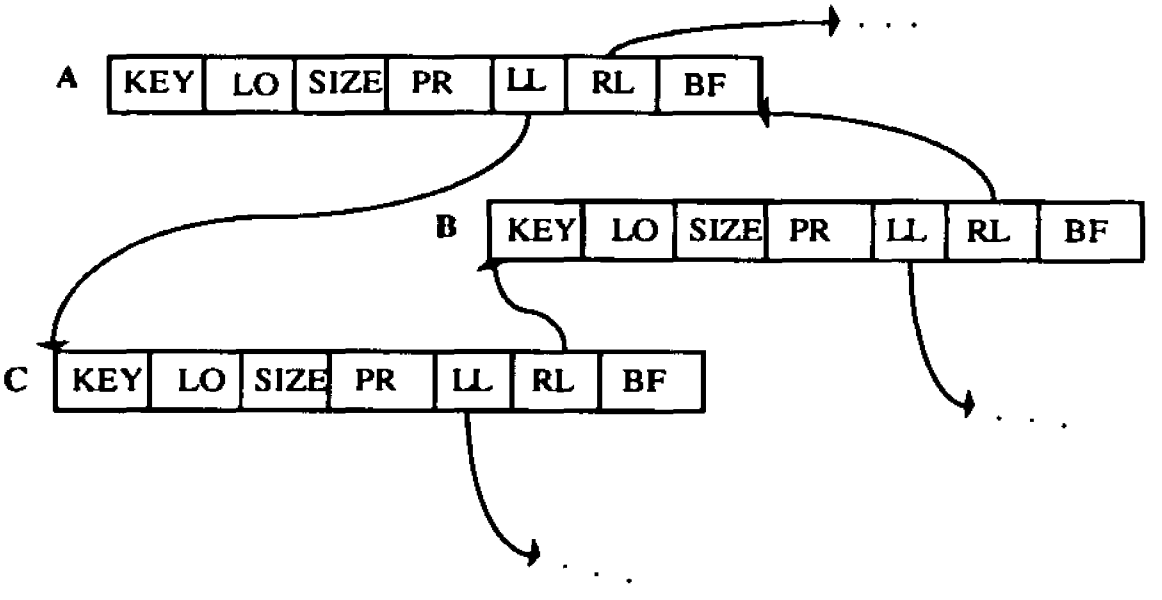


Figure 3.3 Multilist representations. a) Typical multilist representation. b) Schematic representation of a). Each node for both representations has two links which allow one to compute not only left child and right child but also parent node position.

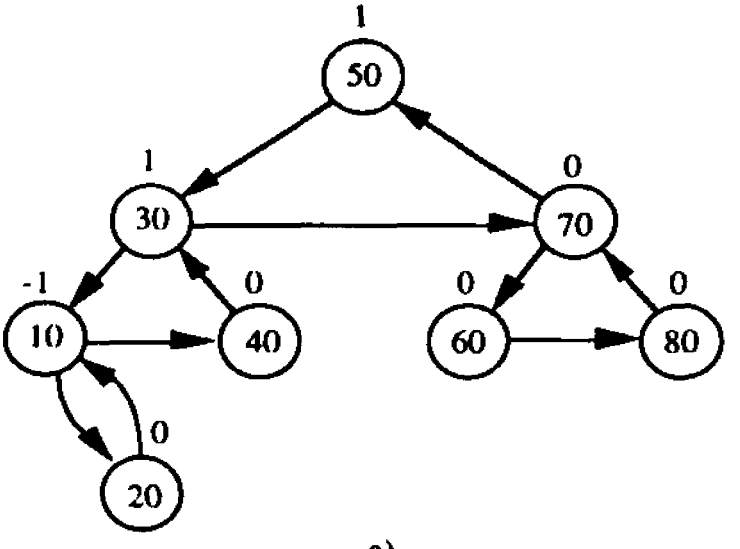


a)



b)

Figure 3.4 Multilist representation of a height balanced tree and its *wst* form. a) A multilist representation of a height balanced tree. The number above each circle is the balance factor as shown in the Figure 3.2. This representation is space efficient as it can express the left child and the right child as well as the parent node using only two links. b) A weighted search tree using a multilist representation. LLINK and RLINK indicate the next available nodes, and the order of the columns are maintained by a heap structure.



a)

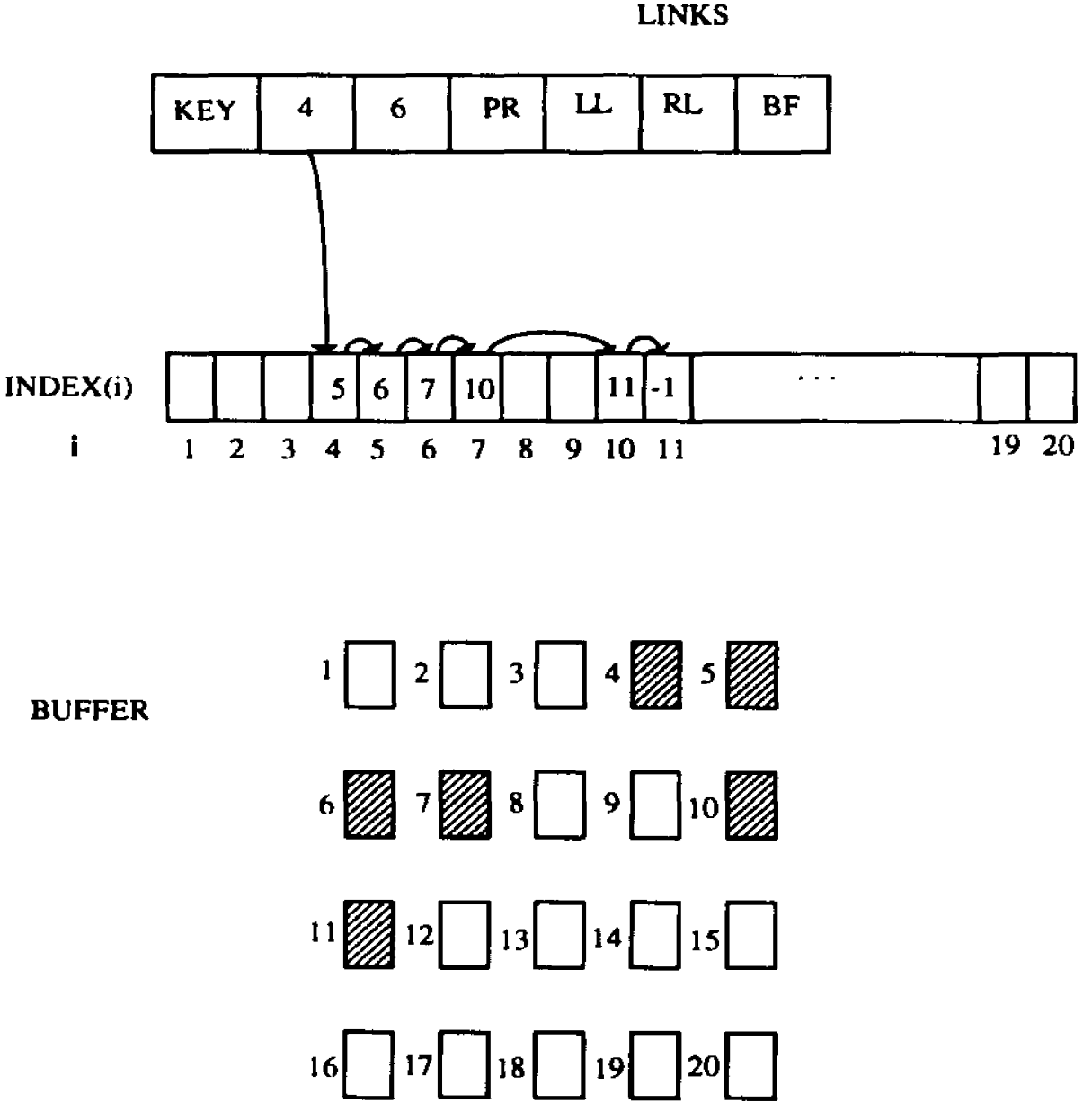
PRIORITY	1	2	5	3	4	7	10	12
KEY	60	40	30	70	10	20	80	50
LLINK	-1	-1	5	1	6	-1	-1	3
RLINK	7	3	4	8	2	5	4	-1
	1	2	3	4	5	6	7	8

b)

When the database is full, which occurs when either the tree or associated storage arrays have reached their maximum capacity, one or perhaps even several elements, depending on the size of incoming data set, have to be deleted before the new information can be stored in the database. Of course, the delete option should only be exercised if the incoming element has a higher priority than the lowest priority element in the database. The delete process checks to see if the free space generated by deleting the lowest priority node suffices to accommodate the incoming data. If it does not, additional nodes are deleted, provided of course that their priorities are less than that of the incoming node, until sufficient free space is obtained. Whenever the delete option is invoked, it is always the lowest priority node that is deleted from the database. The ordering of nodes in the tree according to priority is maintained by the heap structure (see Figure 3.4b). Specifically, the ordering is kept such that the priority of the i-th node is always lower than that of the $2 \times i$-th and $2 \times i + 1$-th nodes where $1 \leq i \leq \lfloor n/2 \rfloor$ and n is the maximum number of nodes in the tree [Lip 86]. The first node always has the lowest priority and is therefore the one that is deleted when the delete option is exercised..

The database system introduced here functions like a management system for disk files. Specifically, the system handles the bookkeeping for storing data (files) of various lengths in (on) a fixed size array (disk). Data are stored in the array as linked allocations [Pet 85] with the *wst* (file directory algorithm) managing the space. As shown in Figure 3.5, the location of information associated with a node in the *wst* is determined by two data elements stored with each node, a data location or head pointer, which is an index that points to the location in a BUFFER array where the first element is

Figure 3.5 Data allocation. The nodes in a *wst* can be used as a directory system. The two data entries LO and SZ that follow the key indicate the location and size of data associated with the node. In the example shown LO = 4 and SZ = 6. The elements in LLBUFF point to blocks in BUFFER that are associated with the node. Given LO and SZ, one can retrieve all the data in BUFFER by using LLBUFF.



actually stored and to the location in the associated link-list array called LLBUFF that holds the pointer for the second element, etc., and a counter size, which specifies the number of fixed size blocks in storage associated with the node. For example, in Figure 3.5, the head pointer is 4 and the size of the data set is 6. The 4 is the index of BUFFER where the first element is found as well as the index of LLBUFF which specifies the location of the next element, LLBUFF(4) = 5, and so on. Hence, as indicated, the 6 date elements associated with the node are located in BUFFER(4), BUFFER(5), BUFFER(6), BUFFER(7), BUFFER(10), and BUFFER(11). Note that there is a redundancy that is built in as LLBUFF(11) = −1 indicates that there are no additional elements associated with the node. This redundancy is necessary to achieve efficient management of the system. If the size of a block is not known, element by element checking would be necessary to determine, for example, the amount of free space that could be gained by deleting it from the database.

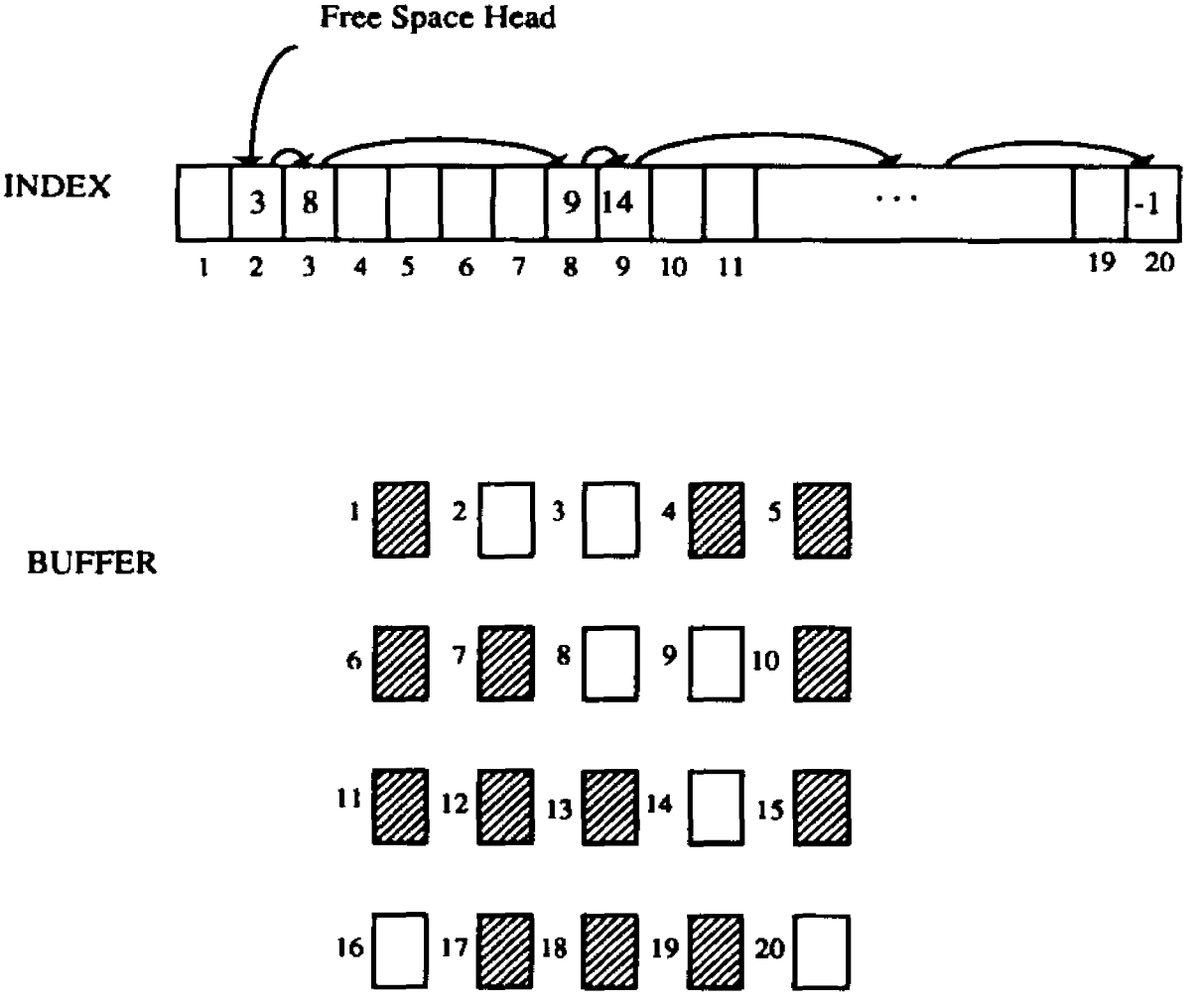
For applications that have a single or fixed number of elements associated with each node, the redundancy referred to above can be removed without any loss as only the first of the two data elements is needed to maintain the structure of the database. In fact, for the fixed-buffer size scenario the link-list array is unnecessary as only fixed-length blocks of data are stored in the buffer and only the starting address or location is required for retrieving any specific one. As this can lead to considerable savings, it is handled as a separate special case in the WSTREE package. A further simplification can be realized if the stored information is fixed-length integer data as it can then be stored directly in the tree as additional data elements per node, a situation that even eliminates the need for keeping a data location index. As the latter is a

very trivial modification of the fixed-buffer size scenario it is not treated separately from it in the WSTREE package. In the discussion that follows, the general case is presented with comments regarding special cases only given when the simplifications on the general results are not straightforward. For the management of free space, the free blocks in the buffer are linked together as shown in Figure 3.6, with a link-list specification using the first three entries of LLBUFF in the same way as for the storage of data. This is illustrated with the help of Figure 3.5. The free space tail index, the head index, and the counter are stored in LLBUFF(-2), LLBUFF(-1), and LLBUFF(0), respectively, for both the fixed-buffer and mixed-buffer size cases.

3.2 Structure of the Weighted Search Tree

The weighted search tree [Par 90c] is a linear integer array, ID(-10:*). The first eleven elements, ID(-10) to ID(0), are reserved for specifying the structure of the tree and other linkage and status information. A full explanation of these entries is given below. The rest of the array, starting with ID(1), is for the node elements. Each node contains four pieces of information: a key that serves to identify the node, integer data that includes the location and size of auxiliary data set associated with the node, the node priority and balance factor, and the linkage instructions. A detailed explanation of the function of each of the four node elements will now be given.

Figure 3.6 Free space management. The LLBUFF array is also used for the management of free space. If the white blocks in BUFFER are free space, then the elements in LLBUFF that point to the free space in BUFFER are linked together as shown in the top diagram. The free space head, the free space tail, and the size of free space are kept in LLBUFF(-2), LLBUFF(-1), and LLBUFF(0), respectively, for later use.



Key -

The key serves as a unique node identifier. It can be a single integer word (simple key) or several integer words (compound key). An allowance for compound keys is necessary because in many applications the number of required labels exceeds the number that can be packed into a single integer. The key is the complete set of integer labels. For a data structure with compound keys the search for a node is carried out by comparing successive key elements, with the left-most one first, the second integer label from the left next, etc. Because the *wst* is height balanced, the search operation can be done in optimal $O(\log n)$ time, where n is the number of nodes in the tree, even when compound keys are employed.

Data -

Information can be stored directly in the tree as integer data, provided the number of such items per node is fixed. Two data elements are reserved for specifying the location and size of information stored in the auxiliary buffer array. This suffices because the data allocation type used for the auxiliary buffer is a linked list so specifying the head of the list and the total number of elements in it is sufficient to retrieved all members. Specifically, if the auxiliary array is called BUFFER and I is the head pointer stored in the *wst* as the node location element, then the first data element is BUFFER(I) and the second and all subsequent elements are also found in BUFFER(I) but with the simple substitution I ←

LLBUFF(I) where LLBUFF is an integer array that maintains the linked list. The number of reserved data elements can be reduced to one and the need for a link-list array eliminated when the information stored in the buffer has a fixed-length rather than a mixed length. Therefore the WSTREE package treats the fixed-buffer and mixed-buffer length cases separately, as we shall discuss below, since the former can be handled with less overhead and therefore with algorithms that execute more efficiently.

Priority and balance factor -

The priority and balance factor are packed into a single integer word. Since the balance factor (BF) requires only two bits, the first two left-most bits in an integer word are used for it and the other 30 bits are used for the priority. The priority scheme consists of two parts: a base priority which reflects the initial value of the node and is under user control and the hit frequency which measures the frequency of use of the node information. The scheme employed in the sample program is a simple one: the first 8 right-most bits are reserved for the base priority and the remaining 22 bits for the hit-frequency times the base priority. The actual value assigned for the priority is therefore the hit frequency times the base priority plus the base priority. For example, if a particular data item has a base priority of 7 and a current hit frequency of 16 then the current assigned priority value is simply 16 x 7 x 256 + 7 = 28679. To decode the priority and balance factor information the intrinsic functions ISHIFT and

IAND can be used [IBM 87]. For example, IAND(I,Z3FFFFFFF) can be used on and IBM system to extract the priority from I and ISHIFT(I,-30) gives the balance factor. In the *wst* a balance factor of 0 is used to indicate that the heights of left-subtree and right-subtree are equal, while if it is 2 the height of left-subtree is one greater than that of the right-subtree, and if it is 3 the height of left-subtree is one less than that of the right-subtree. To encode the priority IP and balance factor IB into a single integer I, the elementary function ISHIFT(IB,30) is added to IP, I = ISHIFT(IB,30) + IP.

Links -

The number of elements dedicated to bookkeeping information in each node is fixed. For the published version of the code this number is two, one for the left link (LL) pointer and one for the right link (RL) pointer. A schematic diagram that illustrates the linkage is given in Figure 3.3. Further details regarding the the *wst* logic will not be given here. Readers interested in that aspect of the problem are referred to Ref. [Par 90c]. The theoretical maximum number of nodes a tree can have is 2^n-1 where n is the number of bits assigned to each pointer. Since this is a large number if the pointers are full integer words, one can economize, for example, by packing the bookkeeping information into one integer word.

The first eleven elements of the tree array, ID(-10) to ID(0), are parameters that specify its structure and provide other information associated with the fetch, insert and delete operations:

ID(-10) = number of nodes currently in the tree;

ID(-9) = maximum number of nodes in the tree;

ID(-8) = location of the parent node of ID(-7);

ID(-7) = location of the node to be balanced;

ID(-6) = location of the parent node at ID(-5);

ID(-5) = location of the current node;

ID(-4) = number of integer words per node for the key;

ID(-3) = number of integer words per node for the key and data;

ID(-2) = ID(-3)+1, location of the priority and balance factor in a node;

ID(-1) = location of the next available node in the tree array;

ID(0) = root node pointer (negative one for a null tree).

As pointed out above, the number of integer data elements per node is at least two, one to specify the location of the first element in the auxiliary buffer array and the other to give its size. However, additional integer data can also be stored in the tree. The only restriction is that the number of integer elements per node used for data storage must be fixed. Variable length data sets, whether integer or real, can only be accommodated through the use of an auxiliary linked-list array. The most economical use of the database system is to place as much information as possible directly into the tree as fixed-length integer data. The use of an auxiliary linked-list array, while entirely feasible and usually necessary, involves more overhead than direct storage.

3.3 Algorithms in the Package

The WSTREE package consists of six main subprograms: TSET, TCHK, TADD, TINS, TDEL, TOUT. In a typical user application, direct calls to only four of these six subroutines, TSET, TCHK, TADD and TOUT, are made. However, TADD uses two additional subroutines, TINS and TDEL, for inserting and deleting nodes from the *wst*. The function of each of these programs is described in this section. A flow chart for a typical application using the WSTREE routines is given in Figure 3.7. It is important to note that the routines are generic and passive. They are generic because they can be used for one or more trees in an application with each tree having a different structure. And they are passive because they work on elements in the tree arrays and need no information other than that provided by the tree about itself. For example, there could be five fetch/insert operations on a tree ID1 and then two on a different one called ID2 followed by seven more on ID1, etc., all without resetting or reinitializing anything. Details about each of the five WSTREE routines follows.

Before an array can be used as a tree, it must be initialized. This is done with a call to the subprogram TSET which has two entries, one for the fixed-buffer {TSETFB} length case and another for the more general mixed-buffer {TSETMB} length scenario:

TSET ···

entry TSETFB(ID,MXNODE,NKEY,NDAT,LLBUFF,MXBUFF)

entry TSETMB(ID,MXNODE,NKEY,NDAT,LLBUFF,MXBUFF)

ID = linear integer array that is to be used as a tree, ID(-10:MXSIZE) where MXSIZE = (MXNODE+1)×(NKEY + NDAT + 3);

MXNODE = maximum number of nodes the *wst* array will accommodate;

NKEY = number of integer words per node dedicated to the key;

NDAT = number of integer words per node dedicated to the data;

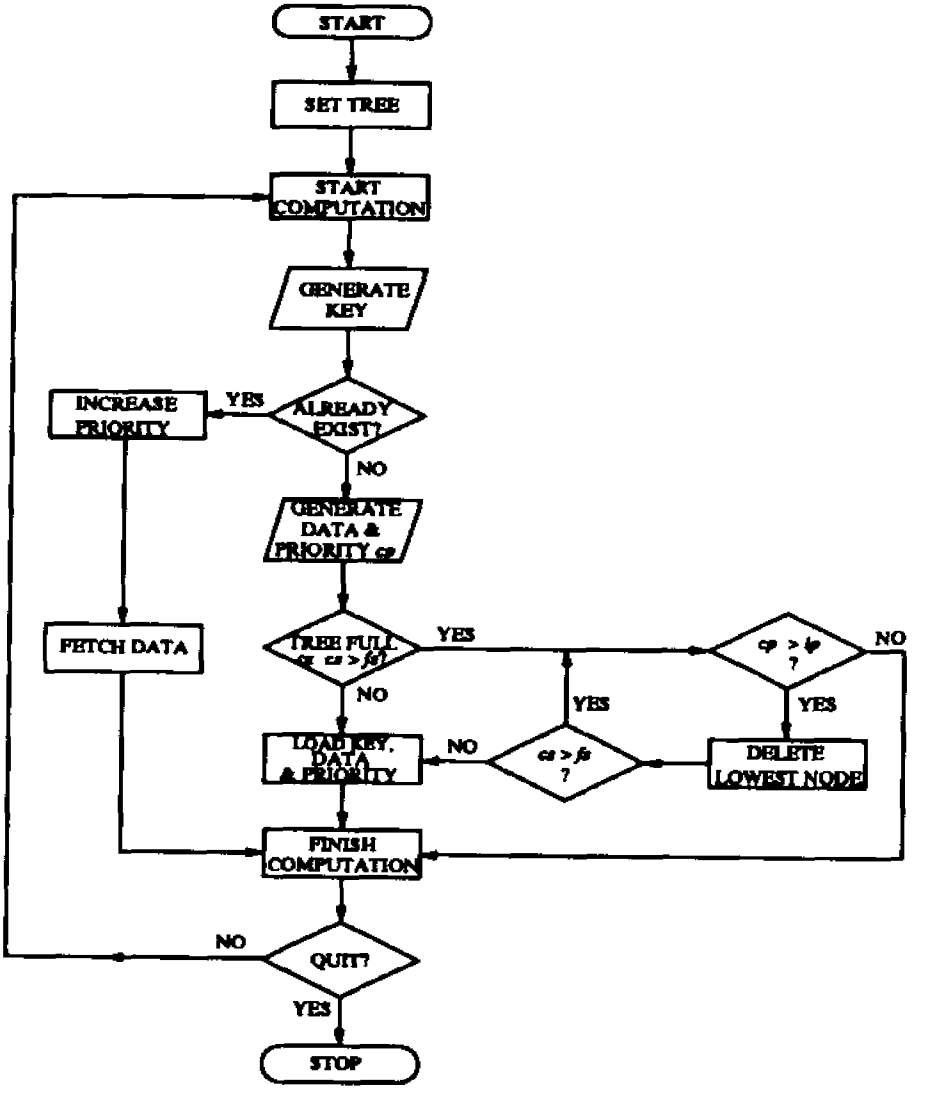
LLBUFF = linear integer array that holds pointer information for the buffer, LLBUFF(-2:0) {FB case} or LLBUFF(-2:MXBUFF) {MB case};

MXBUFF = size of the linear buffer array where information is stored, that is, BUFFER(1:MXBUFF).

In addition to the obvious assignments (see the definitions for ID(-10) to ID(0) given above), a call to TSET initializes the tree by setting ID(-10), ID(-5) and ID(-1) to zero and ID(-8), ID(-7), ID(-6) and ID(0), the root node pointer, to minus one. This value for ID(0) is also a flag that serves to indicate an empty or null tree. Subsequent calls to TSET reinitialize the tree array to null status. Whenever a tree is initialized the information stored in it can no longer be accessed. Indeed, as a precautionary measure, TSET zeros out the tree array: ID(1) = 0, ..., ID(N) = 0.

As explained above, the integer array LLBUFF holds the pointer information for locations in the array BUFFER where the information associated with each node is stored. This link-list array is dimensioned LLBUFF(-2:MXBUFF) with LLBUFF(1) to LLBUFF(MXBUFF) used for pointers to actual data elements in BUFFER and LLBUFF(-2) to LLBUFF(0)

Figure 3.7 Flow chart for a typical application using the WSTREE routines. After a KEY is generated, TCHK is called to see if the key already exists in the database. If the answer is yes, the priority is increased and data retrieved. If the answer is no, the insert procedure is continued. Before an item can be inserted, the tree and free space in the buffer must be checked to see if both are sufficient for the incoming data. If the answer is yes, the incoming record will be inserted (TINS). If the answer is no, items will be deleted from the database, provided their priorities are lower than that of the incoming record, until there is sufficient space to accommodate the new record.



used for information on the free space in BUFFER. Specifically, LLBUFF(-2) is the head of free space, LLBUFF(-1) is the tail of free space, and LLBUFF(0) is the size of free space. A call to TSET sets LLBUFF(-2) to 1, and LLBUFF(-1) and LLBUFF(0) to MXBUFF.

Before describing the subprograms TCHK, TADD, TINS and TDEL, the algorithms for computing the positions of the left child and the right child of a particular node need to be introduced. To make the WSTREE package execute efficiently these algorithms have not been separated out as subroutines in the WSTREE package, nonetheless, for convenience they are identified here as if they are subroutines called LCHILD and RCHILD, respectively. The return value p in LCHILD and RCHILD are the left child position and the right child position, respectively.

LCHILD(p)

```
if LLINK(p) = nil then
p ← nil
elseif RLINK(LLINK(p)) = p then
if BF(p) = -1 then
        p ← LLINK(p)
    else
        p ← nil
else
    p ← LLINK(p)
```

RCHILD(p)

```
if LLINK(p) = nil then
    p ← nil
elseif RLINK(LLINK(p)) = p then
    if BF(p) = -1 then
        p ← nil
    else
        p ← LLINK(p)
else
    p ← RLINK(LLINK(p))
```

The subprogram TCHK is used for fetch operations. A call to TCHK effects a search of ID to see if it contains a node with a key equal to NKEY. If it does, the program executes a RETURN 1 after increasing the priority of the node that was found by an amount equal to 256 times the base priority and reconstructing the heap. If NKEY is not found, the program executes a normal RETURN and ID(-5) is assigned information on where to insert NKEY into the *wst* when added via a call to TINS. A schematic of the logic used in TCHK will now be given. The element being sought is x and t refers to the root node:

TCHK(NKEY,ID,*)

NKEY = linear integer array containing the key of the incoming item;

ID = linear integer array that is being used for the tree.

Step 1. Check for a null tree:

if t = nil **then**

goto Step 3

Step 2. Compare:

set $p \leftarrow t$

while $p \neq$ nil **do** Steps 2.1 - 2.3:

Step 2.1 **if** *KEY[p] > KEY[x]* **then** $t \leftarrow p$; call LCHILD(p)

Step 2.2 **elseif** *KEY[p] < KEY[x]* **then** $t \leftarrow p$, call RCHILD(p)

Step 2.3 **else**

increase the priority of node t;

reconstruct the heap;

RETURN 1

endwhile

Step 3. RETURN (normal)

The subroutine TADD is used to add an element to an existing database provided space is available. As with TSET, there are two entries, one for the fixed-buffer length scenario and another for the mixed-buffer length case. If space is not available and the priority of the incoming element is greater than the lowest priority element currently in the database that element is eliminated to make room for the new one. The last scenario is repeated as necessary and appropriate in an attempt to accommodate the new element. During the execution of the TADD procedure calls to the subroutines TDEL for deleting the lowest priority node and TINS for inserting the incoming node in

the tree are made. These two subroutines are described in the below. The logic of the subroutine TADD is the following:

TADD ···

entry TADDFB(NKEY,NDAT,BULOAD,NOSIZE,NPBASE,ID,BUFFER,LLBUFF)

entry TADDMB(NKEY,NDAT,BULOAD,NOSIZE,NPBASE,ID,BUFFER,LLBUFF)

NKEY	=	linear integer array containing the key of the incoming item;
NDAT	=	linear integer array containing the data of the incoming item;
BULOAD	=	linear real array were the incoming data elements are stored;
NOSIZE	=	size of the incoming data item;
NPBASE	=	base priority that is to be assigned to the incoming item;
ID	=	linear integer array that is being used for the tree;
BUFFER	=	linear real array where the BULOAD elements are to be stored;
LLBUFF	=	linear integer array that hold pointer information for BUFF.

Step 1. Check if either the tree or buffer is full:

if yes, check if the priority of the incoming item is higher than that of lowest priority node in the tree

if yes, delete lowest priority node (*call TDEL*)

goto Step 1

else goto Step 3

Step 2. Insert the incoming item into the database (*call TINS*)

Step 3. RETURN

Fixed-length integer and mixed-length real data can be stored in the database without any program modification. The integer data can be inserted directly into the tree as additional data elements. If the real data is also fixed in length it can be stored in the auxiliary buffer array without the use of a link-list indexing scheme. This is accomplished by using the TSETFB and TADDFB rather than the TSETMB and TADDMB subprograms. Variable length integer data can also be accommodated if the user changes the real array called BUFFER in TADD to an integer array, say INTGER. Likewise, complex data can be stored by again simply changing BUFFER in TADD to a complex array, say CMPLEX. If no integer data is to be stored in the tree then for the general case NDAT=2 and for the fixed-buffer size scenario NDAT=1. If the only data is fixed-length integer data even the buffer is not needed. This can be handled in a trivial manner by simply setting MXBUFF=1. A better solution, however, is to modify the program by eliminating all references to the buffer in TSETFB and TADDFB, including the use of the first and second data element for the specifying the location and size of data in the auxiliary buffer. In some applications it may be desirable to have more than one auxiliary set of data associated with a single *wst* structure. This too can be easily achieved but requires additional straightforward modifications of the TSET and TADD subprograms. A very important feature is that the four routines TCHK, TINS, TDEL and TOUT, which involve rather complex manipulations of the *wst*, can be used for all scenarios without any modification.

The subroutine TINS is used to insert a node into the *wst*. TINS must be preceded by a call to TCHK which checks to see whether or not the item NKEY is already in the *wst*. After inserting NKEY into the *wst*, TINS reconstructs the heap.

TINS(NKEY,ID)

NKEY = linear integer array containing the key of the incoming item;
ID = linear integer array that is being used for the tree.

Step 1. Insert new node into the tree:
if tree is empty then
Set $p \leftarrow x$, left child of $x \leftarrow$ nil, right child of $y \leftarrow$ nil

Step 2. Balance the tree:
If height difference > 1 rotate the tree, reset the tree

Step 3. Reconstruct the heap

Step 4. RETURN (normal)

The subroutine TDEL deletes a node from a *wst*. This is a necessary feature for an fixed-space, on-line system since a situation can be encountered where either the tree or the buffer where real data is stored is full, and a node with a higher priority than the lowest priority node in the tree needs to be added to the *wst*. The subroutine TDEL can be executed to make space for the incoming item.

TDEL(ID)

ID = linear integer array that is being used for the tree.

Step 1. Find x along a path from the root node to node y:

$p \leftarrow ROOT$;

while ($p \neq y$) **do**

if (p satisfies one of the conditions for x) **then** $x \leftarrow p$;

if ($KEY[p] > KEY[y]$) **then** call LCHILD(p);

else if ($KEY[p] < KEY[y]$) **then** call RCHILD(p);

endwhile

Step 2. Delete node y in the linear tree array:

if (*left child of* y = *null*) **then**

if (*right child of* $y \neq$ *null*) **then** call RCHILD(y)

else $y \leftarrow$ *null*

else if (*right child of* y = *null*) **then** call LCHILD(y)

else begin

$z \leftarrow$ the node contains the largest key value smaller than $KEY[y]$;

$i \leftarrow y$;

while ($i \neq$ *null*) **do**

if (i satisfies one of the conditions for x) **then** $x \leftarrow i$;

if ($KEY[i] > KEY[z]$) **then** call LCHILD(i);

else if ($KEY[i] < KEY[z]$) **then** call RCHILD(i);

endwhile

$y \leftarrow z$;

end

Step 3. Balance the tree:

while (x is not a leaf) **do**

if (height difference of subtrees of $x > 1$) **then** *rotate*;

if ($KEY[y] < KEY[x]$) **then** call RCHILD(x);

else call LCHILD(x);

endwhile

Step 4. Reconstruct the heap

Step 5. RETURN (normal)

The subroutine TOUT can be used to traverse *wst* to find specific nodes:

TOUT(NFILE,NAD,MIN,MAX,NSTEP,ID);

NFILE	=	file number of the output device where node information is to be written; if NFILE = 0 output is suppressed;
NAD	=	1 if the traversal is to be in traversed in ascending order, otherwise it will be traversed in descending order;
MIN	=	number of the first node that is to be retrieved;
MAX	=	number of the final node that is to be retrieved;
NSTEP	=	step size; nodes that are integer multiples of NSTEP beyond MIN up to MAX will be retrieved;
ID	=	name of the tree array that is to be traversed.

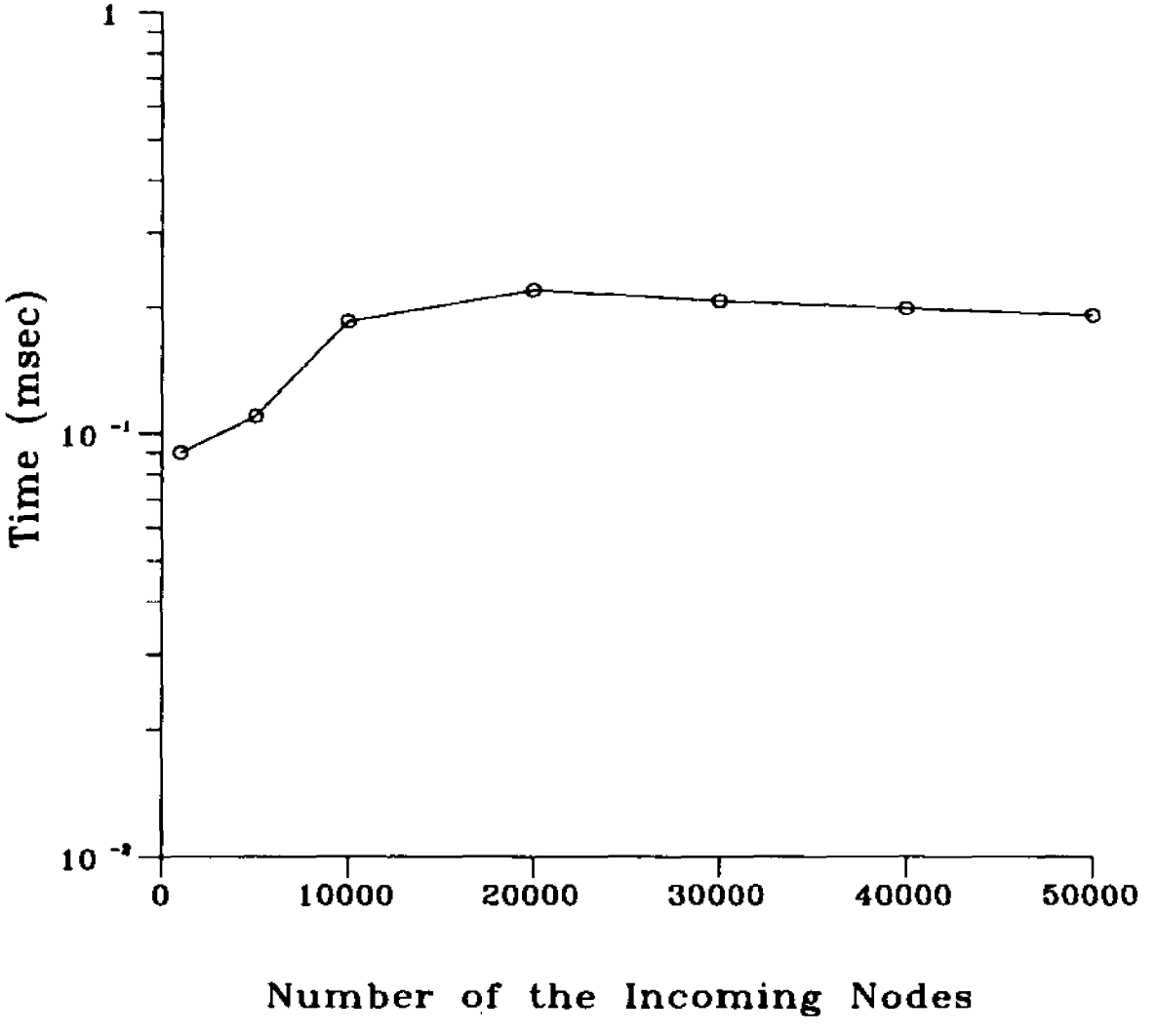
The node number, key, and data are written out (NFILE ≠ 0) using the following fixed format: (1X, I6, ':', 2X, 14I5/(10X,14I5)).

3.4 Performance Characteristics

To test the overall performance of the WSTREE program, a sample program was written that used a simple routine called IRAND and ISIZE for generating integer random numbers between 1 to 255 for the key and priority elements of the nodes and numbers between 1 to 5 for the number of data elements associated with each node and stored in BUFFER. The size of BUFFER was set at 20,000 integer words. For testing purposes, the number of integer words used for the key was fixed at 5. Hence the number of integer words per node was 10: 5 for the key, 2 for the location and size parameters, 1 for the priority and balance factor, and 2 for the links. The average time required for fetch, insert and delete operations using WSTREE on this data structure with between 1000 and 50,000 nodes was determined using system timing routines. The results for execution on an IBM 3090/600E is shown in Figure 3.8. The average time per node for the generation of a 50,000 item set stored in a 20,000 item *wst* tree with an associated 20,000 item buffer array was 0.19 msec for the mixed-buffer size case and 0.16 for the corresponding fixed-buffer size scenario. When the situation allows, the use of the fixed-buffer size routines should therefore be expected to yield gains of about 16% in execution speed over use of the general mixed-buffer size programs.

Another test was made to demonstrate just how the WSTREE routine works. For this test, which also used the IRAND and ISIZE routines, the size

Figure 3.8 Performance test for the *wst*. Average time for fetch, insert and delete operations on a sample database are plotted as a function of the number of the incoming nodes. Results shown are for a test run, described in the text, that used the routines in the WSTREE package on an IBM 3090/600E mainframe computer. System timing routines were used in the test.

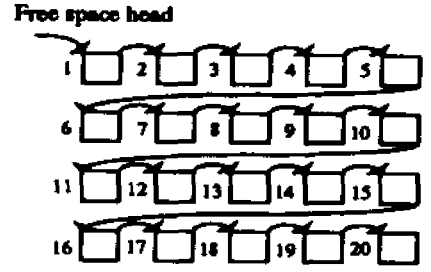


of the fixed storage BUFFER was set at 20 integer words. Figure 3.9 shows the evolution of the database through the addition of 7 elements. Note that to accommodate the fifth element generated, the third had to be deleted and to accommodate the seventh the fourth had to go. Clearly, as with a file directory system, the storage area can quickly become fragmented. Should this be a problem, the database can be defragment by writing the information out to disk using TOUT and then reloading the database using TINS as it is read back in. Since all of the routines in the WSTREE package are time-optimal, the disk input-output is the slow part of this procedure. But this too can be avoided if rather than writing the output from TOUT to disk it is used directly by TINS to load another duplicate database. Though faster, this procedure would require double the space. A breakdown on the storage requirements for the routines in WSTREE on an IBM 3090/600E is given in Table 1.

3.5 Discussion

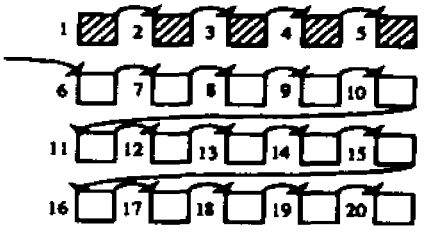
The routines in the WSTREE package are simple to use and execute efficiently. In applications where the number of redundant calls to a subprogram is large and the time spent in that subroutine is a sizable fraction of the total CPU time, the use of a *wst* for saving intermediate results and thereby reducing the redundancy can result in major gains. An important feature of the WSTREE database system is that, while in many ways it is like a file directory system, it goes beyond this because nodes can be assigned priorities. Furthermore, when the assigned storage space is used up, there is a delete mechanism available

Figure 3.9 Schematic diagram for the WSTREE as a directory system. The diagrams on the left side are for the BUFFER and those on the right side are the directory entries stored in the *wst*. The sequence from a) to h) shows how the WSTREE routines work as a directory system in a fixed size storage environment. For demonstration purposes, an array BUFFER of 20 integer words was selected. The crosshatched blocks on the left side that are occupied by data elements and those that are white are free space.



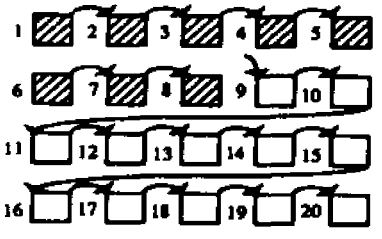
Empty Directory

a)



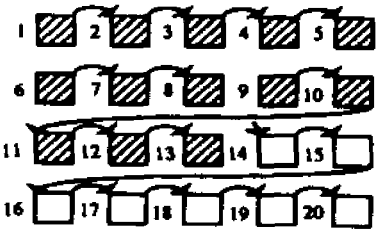
KEY	PR	LO	SZ
K1	109	1	5

b)



KEY	PR	LO	SZ
K1	109	1	5
K2	144	6	3

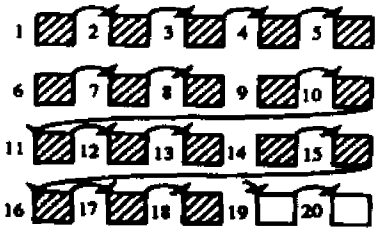
c)



KEY	PR	LO	SZ
K1	109	1	5
K2	144	6	3
K3	37	9	5

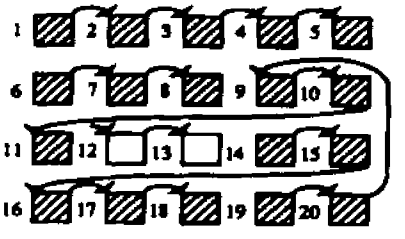
d)

continued



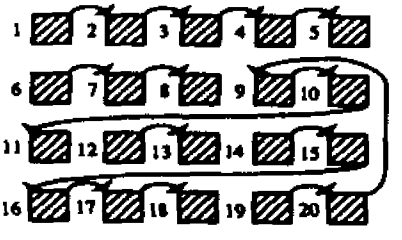
KEY	PR	LO	SZ
K1	109	1	5
K2	144	6	3
K3	37	9	5
K4	58	14	5

e)



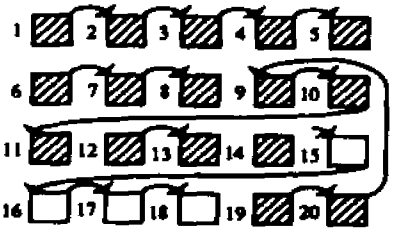
KEY	PR	LO	SZ
K1	109	1	5
K2	144	6	3
K4	58	14	5
K5	158	19	5

f)



KEY	PR	LO	SZ
K1	109	1	5
K2	144	6	3
K4	58	14	5
K5	158	19	5
K6	181	12	2

g)



KEY	PR	LO	SZ
K1	109	1	5
K2	144	6	3
K5	158	19	5
K6	181	12	2
K7	159	14	1

h)

that allows the user to keep the most valuable information in the database at the expense of throwing out the least valuable.

The routines in the WSTREE package have been written in a manner that makes incorporating them into existing programs an easy task. In particular, the codes are written in FORTRAN as, despite attempts to adopt simpler and more powerful languages, this is still the language of choice for most CPU-intensive scientific applications. One reason for this is the ready available of many FORTRAN codes that have been tuned for efficiency, which is particularly important in high-performance computing scientific applications. Another reason is that the efficiency of FORTRAN compilers is generally very high.

The purpose of the WSTREE package is to place in the public domain a set of software tools that allows scientists to incorporate database structures and logic into FORTRAN programs with a minimum of inconvenience. We have found the WSTREE routines extremely useful in numerically intensive applications. It is our hope that others will also.

Table 3.1 Storage requirements in bytes for routines in the WSTREE package on an IBM 3090/600E. Numbers in parenthesis include array storage for 2500 nodes in the tree and a fixed size buffer and pointer arrays of 5000 words each.

TSET	972
TCHK	3072
TADD	1908
TINS	4952
TDEL	7944
TOUT	1876
TOTALS(WSTREE)	20724
IRAND	442
ISIZE	442
DRIVER	2944 (142944)
TOTALS(ALL)	24552 (163668)

CHAPTER 4
NEW REPESENTATION OF A SPARSE MATRIX

So far the discussion has been restricted to a data structure, called a weighted search tree (*wst*), and its application in the development of a numerical database system that can be used to make scientific computations more efficient by reducing the number of redundant calculations. Another way to improve performance is to develop more efficient algorithms. In this chapter, a new representation of a sparse matrix is introduced that is very efficient for matrix multiplication when the nonzero elements are partially or fully adjacent to one another as in band or triangular matrices. Space complexity of the new representation is better than that of existing algorithms when the number of the groups of adjacent nonzero elements is less than two-thirds of the total number of nonzero elements. The time complexity is also better or even much better than that of existing algorithms depending on the number of groups of nonzero adjacent elements in the factor matrices.

4.1 Introduction

Matrices are abstract mathematical objects that arise in many numerical as well as non-numerical problems. Examples range from solving systems of equations to representations of graphs. In large-scale, high-performance scientific and engineering applications, it is not unusual to encounter matrices with tens of thousands of elements. Designing efficient

algorithms for matrix operations is therefore a problem of fundamental importance in computational science.

Consider the multiplication of an $m \times n$ matrix A and an $n \times p$ matrix B. If A and B are represented by two-dimensional arrays, the simplest method is as follows:

```
For i =1 to m do
    For j =1 to p do
        C_ij = 0
        For k = 1 to n do
            C_ij = C_ij + A_ik × B_kj
        endfor
    endfor
endfor
```

Obviously this algorithm takes $O(n^3)$ time if $m = n = p$. This procedure will be called the *standard matrix multiplication algorithm*. A trivial lower bound on the time complexity for multiplying two general $n \times n$ matrices is $\Omega(n^2)$. Several algorithms with $O(n^x)$ running time, where $2 < x < 3$, have been proposed [Sta 86]. However, most of these algorithms are impractical because the constant factors associated with their time complexities are large. The design and implementation of algorithms that yield an improved lower and upper bound for the general matrix multiplication problem therefore remains an outstanding open challenge.

It is important for computer scientists to understand the special requirements of large-scale matrix applications and design algorithms that meet these needs. These requirements can be identified by considering the common characteristics of abstract mathematical models of the physical world. In this chapter the multiplication of large sparse matrices [Bur 85, Cul 85, Gol 89], which enter for example in eigensystem analyses [Neg 88], least square problems [Bet 89], and solutions to systems of linear equations, will be considered. For sparse matrix multiplication, the lower bound mentioned above for the time complexity certainly does not hold. A trivial example of this is when one of the factor matrices is diagonal in which case the time complexity is simply $O(n)$. It is therefore desirable to design special sparse matrix algorithms with time and space complexities that depend on the number of nonzero elements in the operand matrices.

A space efficient representation of sparse matrices is given in [Hor 83]. In this representation only nonzero elements are considered with each identified by a 3-tuple of the form (i, j, v), where i and j are the row and column numbers of the element, respectively, and v is the value of the element. A sparse matrix is stored in memory as a linear list of nonzero elements in row-major order. With this representation, multiplying two $n \times n$ matrices can be carried out in $O(n\ (t_1 + t_2))$ time, where t_1 and t_2 are the numbers of nonzero elements in the operand matrices.

In this chapter, a modified data structure for sparse matrices is introduced and a matrix multiplication algorithm based on this new structure is presented. The analysis shows that the new data structure and algorithm are more time and space efficient than the existing methods if the sparse matrices

contain segments consisting of adjacent nonzero elements in rows and/or columns. The improved performance of the algorithm is verified by numerical experiments. In this regard it is important to note that in large-scale, high-performance computing applications, even a factor of two improvement in the time efficiency of an operation may impact on whether a particular application can or cannot be run. Since the sparse matrices arising in scientific and engineering applications tend to be highly structured, such as band or triangular matrices, the implementation of the matrix multiplication algorithm introduced in this chapter can result in a considerable performance improvement.

4.2 Sparse Matrix Representations

It is obvious that using two-dimensional arrays to represent sparse matrices not only wastes space but also cannot lead to sub-quadratic time operations. Consider the sparse matrix A shown in Figure 4.1. The data structure for sparse matrices introduced in [Hor 83] for this example is shown in Figure 4.2.

Let A be represented by a $(t + 1) \times 3$ array LA(0:t, 1:3), where t is the number of nonzero elements in the matrix. The 3-tuple (LA(k, 1), LA(k, 2), LA(k, 3)) specifies the k-th entry of the LA array. The 0-th entry contains global information on A, with LA(0, 1) and LA(0, 2) specifying the number of rows and columns, respectively, and LA(0, 3) the number t of nonzero elements. The k-th entry, $k > 0$, contains information on the k-th nonzero element. Specifically, LA(k, 3) is its value and LA(k, 1) and LA(k, 2) are the

Figure 4.1 Two-dimensional array representation of a 7 × 7 matrix. The standard two-dimensional array representation of a matrix uses fixed storage since zero and nonzero elements are handled the same.

	col 1	*col 2*	*col 3*	*col 4*	*col 5*	*col 6*	*col 7*
row 1	2	0	0	0	0	0	0
row 2	3	4	0	0	0	0	0
row 3	0	0	0	5	0	0	0
row 4	0	0	0	6	0	0	0
row 5	0	1	0	8	4	3	0
row 6	0	0	0	0	2	2	1
row 7	0	0	0	0	5	0	0

Figure 4.2 Horowitz-Sahni representation of a 7 × 7 matrix. The elements LA(0, k) with k = 1, 2 and 3 specify, respectively, the number of rows, columns, and nonzero elements, t, in the matrix A. The elements LA(1:t, k) with k = 1 and 2 specify the row and column indices of the nonzero element of A that is stored in L(1:t, 3). Specifically, LA(i, 1) and LA(i, 2) are the row and column labels of the i-th nonzero element which has the value stored in LA(i, 3), $1 \leq i \leq t$.

	0	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*	*11*	*12*	*13*
1	**7**	**1**	**2**	**2**	**3**	**4**	**5**	**5**	**5**	**5**	**6**	**6**	**6**	**7**
2	**7**	**1**	**1**	**2**	**4**	**4**	**2**	**4**	**5**	**6**	**5**	**6**	**7**	**5**
3	**13**	**2**	**3**	**4**	**5**	**6**	**1**	**8**	**4**	**3**	**2**	**2**	**1**	**5**

row and column indices, respectively. Furthermore, let the nonzero elements be stored in this array in row-major order, that is, in increasing order of the row index, and for all nonzero elements with the same row index in increasing order of the column index. The LA array is a linear list of nonzero elements in the matrix A.

Using this data structure and the Horowitz-Sahni algorithms [Hor 83], the transposition of an $m \times n$ matrix can be done in $O(n + t)$ time, where t is the total number of nonzero elements in the matrix. The multiplication of an $m \times n$ matrix A with an $n \times p$ matrix B can be accomplished in $O(pt_A + mt_B)$ time, where t_A and t_B are the number of nonzero elements in A and B, respectively. It is easy to see that this data structure is not space efficient if the matrix contains segments of adjacent nonzero elements, since most of the row and column numbers in a segment are then redundant. More importantly, this redundant structure does not lead to more efficient matrix operations, which is contrary to the general algorithm design principle of introducing redundancy to improve performance.

The Horowitz-Sahni matrix multiplication algorithm performs repetitive linear scans of the entries of the arrays representing the factor matrices. In computing C_{ij}, the factors A_{ik} and B_{kj} are both checked to see if the multiplication operation $A_{ik} \times B_{kj}$ can be skipped because one or the other factor is zero. It would be nice to add additional information to the data structure that would allow redundant checks to be eliminated. This is the idea behind the new data structure. Specifically, a *maximal nonzero segment* (or simply *segment*) in row i of an $m \times n$ matrix A is defined as a closed interval $[j, k]$ such that all elements $A_{il} \neq 0$ for $j \leq l \leq k$ with $A_{i,j-1} = 0$ if $j > 1$ and $A_{i,k+1} =$

0 if $k < m$. Our representation for A consists of two arrays, EA(1:t_A) and SA(0:s_A, 1:3), where t_A is the number of nonzero elements in A and s_A is the number of segments in A in row-major order. Each segment corresponds to a sub-array of EA and all segments of A are in row-major order in EA. SA contains s_A+1 entries with each entry being a 3-tuple (SA(r, 1), SA(r, 2), SA(r, 3)). SA(0, 1) and SA(0, 2) specify the number of rows and columns in A, respectively, while SA(0,3) gives the number of segments. That is, SA(0, 1) = m, SA(0, 2) = n and SA(0, 3) = s_A for an $m \times n$ matrix with s_A segments. The r-th entry, r > 0, contains information about the r-th segment. Specifically, if the r-th segment $[j, k]$ is in row i, then SA(r, 1) = i, SA(r, 2) = j, and SA(r, 3) = k. Since the segments of A are arranged in EA in a unique linear order, the indices of the segments can be used to calculated a linear scan of SA. We use the pair (EA, SA) to denote this data structure for the matrix A. For the example matrix A of Figure 4.1, this new representation is shown in Figure 4.3.

For an $m \times n$ matrix A with t_A nonzero elements and s_A segments, the conventional 2-dimensional array representation of A requires $m \times n$ words, LA requires $3(t_A + 1)$ words and our representation requires $t_A + 3(s_A + 1)$ words. If A is sparse, then it is possible that $3(t_A + 1) << m \times n$ and $[t_A + 3(s_A + 1)] << m \times n$. However, if $t_A > 1.5\ s_A$, then $[t_A + 3(s_A + 1)] < 3(t_A + 1)$. That is, if on the average each segment contains more than 1.5 elements, then our new representation requires less space than the representation LA introduced in [Hor 83]. For large-scale scientific applications, this condition most probably holds. It is easy to see that $O(t_A)$ time is sufficient to transform LA into (EA, SA), and vice versa. It is a simple fact that transforming

Figure 4.3 New representation of a sparse matrix. This representation consists of two arrays, EA and SA. The value of the nonzero elements are stored in a one-dimensional array EA while the two-dimensional array SA holds the information on the segments of the matrix A. Specifically, while as for the Horowitz-Sahni case SA(0,k) with k = 1 and 2 specify the number of rows and columns in A, SA(0,3) gives the number of segments, t_A, rather than the number of nonzero elements. Furthermore, SA(i,1) specifies the row index of segments while SA(i,2) and SA(i,3) give the starting and ending columns of the i-th segment, $1 \le i \le t_A$.

EA:

1	2	3	4	5	6	7	8	9	10	11	12	13
2	3	4	5	6	1	8	4	3	2	2	1	5

SA:

	0	1	2	3	4	5	6	7	8
1	7	1	2	3	4	5	5	6	7
2	7	1	1	4	4	2	4	5	5
3	8	1	2	4	4	2	6	7	5

a conventional 2-dimensional array representation of an $m \times n$ matrix into LA, or (EA, SA), and vice versa, takes $O(m \times n)$ time.

4.3 Sparse Matrix Multiplication

Consider the problem of multiplying an $m \times n$ matrix A and an $n \times p$ matrix B to obtain an $m \times p$ matrix C using this new matrix representation. Logic similar to the matrix multiplication algorithm given in [Hor 83] can be used provided an efficient matrix transposition algorithm is available. Let (EB, SB) be the new representation matrix B. Then (EB^T, SB^T) can be constructed as follows: first, transform (EB, SB) into LB in $O(t_B)$ time; then transform LB into LB^T in $O(n + t_B)$ time (see [Hor 83]); and finally, transform LB^T into (EB^T, SB^T), again in $O(t_B)$ time. To reduce the overhead, a bucket sort algorithm [Aho 74] can be applied:

TRANSPOSE Algorithm

Step 1. Initialize n empty queues;

Step 2. Scan EB[i] in increasing order of i, use SB to calculate the row and column number of EB[i], and put EB[i] into the j-th queue if it is in the j-th column of B;

Step 3. Scan queues to compute SB^T and concatenate the result to obtain EB^T;

end of TRANSPOSE

This algorithm also takes $O(n + t_B)$ time. However, the constant factor of its complexity is smaller than that of the previous method. The details of TRANSPOSE are given in Appendix B.

Before presenting the matrix multiplication algorithm, it is instructive to first consider the dot product of two vectors $A = (A_1, A_2, ..., A_n)$ and $B = (B_1, B_2, ..., B_n)$ of n elements each. Let A (resp. B) be represented as a sequence of segments of nonzero elements

$$L^A = ([j_1^A, k_1^A], [j_2^A, k_2^A], \cdots, [j_r^A, k_r^A])$$

$$(\text{resp. } L^B = ([j_1^B, k_1^B], [j_2^B, k_2^B], \cdots, [j_s^B, k_s^B])$$

such that

$$1 \le j_1^A \le k_1^A < j_2^A \le k_2^A < \ldots < j_r^A \le k_r^A \le n$$

$$(\text{resp. } 1 \le j_1^B \le k_1^B < j_2^B \le k_2^B < \ldots < j_s^B \le k_s^B \le n)$$

There are a total of nine possible different segment overlap conditions as shown in Figure 4.4. In the figure the left rectangle and right rectangle represent segment $[j_a^A, k_a^A]$ and $[j_b^B, k_b^B]$, respectively. The solid area corresponds to the overlap interval of the two segments. The nine cases can be subdivided into three groups which are labelled (1), (2) and (3) in the figure. Based on the overlap conditions, the following simple algorithm can be given:

VECTOR_MULTIPLY Algorithm

Step 1. Initialize variables;

$a := 1, b := 1, c := 0,$

```
Step 2. while a ≤ r and b ≤ s do
            [j, k] := [j_a^A, k_a^A] ∩ [j_b^B, k_b^B]
            if [j,k] ≠ φ then
               begin
                  compute the dot-product of sub-vectors of A and B defined by
                  segment [j,k], and let D be the result of this computation;
                  C := C + D;
                  if the overlap condition of [j_a^A, k_a^A] ∩ [j_b^B, k_b^B] is one of types
                     1.b and 1.c in Figure 4.4 then
                     a := a + 1
                  else if the overlap condition is one of types 2.b and 2.c then
                     b := b + 1
                  else /* the condition is one of types 3.a, 3.b and 3.c */
                     a := a + 1
                     b := b + 1;
               end
            else
               /* the overlap condition of [j_a^A, k_a^A] ∩ [j_b^B, k_b^B] is one of types
               1.a and 2.a */
                  a := a + 1;
                  b := b + 1;
         endwhile
end of VECTOR_MULTIPLY
```

Define

$$\text{Overlap}_{ab} = \begin{cases} 0, \text{ if segments } [j_a^A, k_a^A] \text{ and } [j_b^B, k_b^B] \text{ do not overlap;} \\ \min\{k_a^A, k_b^B\} - \max\{j_a^A, j_b^B\}, \text{ otherwise.} \end{cases}$$

and

$$\text{Overlap}_{AB} = \sum_{\substack{1 \le a \le r \\ 1 \le b \le s}} \text{Overlap}_{ab}.$$

Clearly, the time complexity of VECTOR_MULTIPLY is $O(r + s + \text{Overlap}_{AB})$. It is also easy to see that $\Omega(\text{Overlap}_{AB})$ is a lower bound for the time complexity of the dot product of two vectors A and B. If r + s and Overlap_{AB} are comparable, then VECTOR_MULTIPLY runs in optimal time. The worst case scenario is when $A = (A_1, \cdots, A_n)$ and $B = (B_1, \cdots, B_n)$ are of the forms $A_i = 0$, $A_{i+1} \neq 0$ & $B_i \neq 0$, $B_{i+1} = 0$ for either i even or i odd. In this case r + s = n and $\text{Overlap}_{AB} = 0$. To summarize, by introducing segment information into the data structure for sparse vectors, non-overlap portions of the two operands can be skipped, and if the two vectors contain many adjacent nonzero elements, the run time of comparison operations for determining overlapping portions of segments will be small.

Based on these observations, the following matrix multiplication algorithm is proposed. As input there is an $m \times n$ matrix A and an $n \times p$ matrix B represented by (EA, SA), and (EB, SB), respectively, while (EC, SC) represents the product matrix C.

Figure 4.4 Overlap conditions for the multiplication of two matrices. In the figure each vertical strip denotes a matrix segment. The segment on the left is from A and the one on the right is from B^T in the product A × B. The solid block denotes the overlap interval. The patterns labelled Types 1, 2 and 3 are complementary scenarios. Specifically, if the current overlap is of Type 1 (2) then the next pair that must be considered is found by increasing the segment index of A (B^T). If the overlap is of Type 3, that is, SA(i,3) and SB(j,3) are equal for the i-th and j-th segments of A and B, respectively, the segment indices of both A and B^T must be increased. The implementation of this sequential procedure yields a time complexity that is proportional to the total number of segments as compared to the number of nonzero elements.

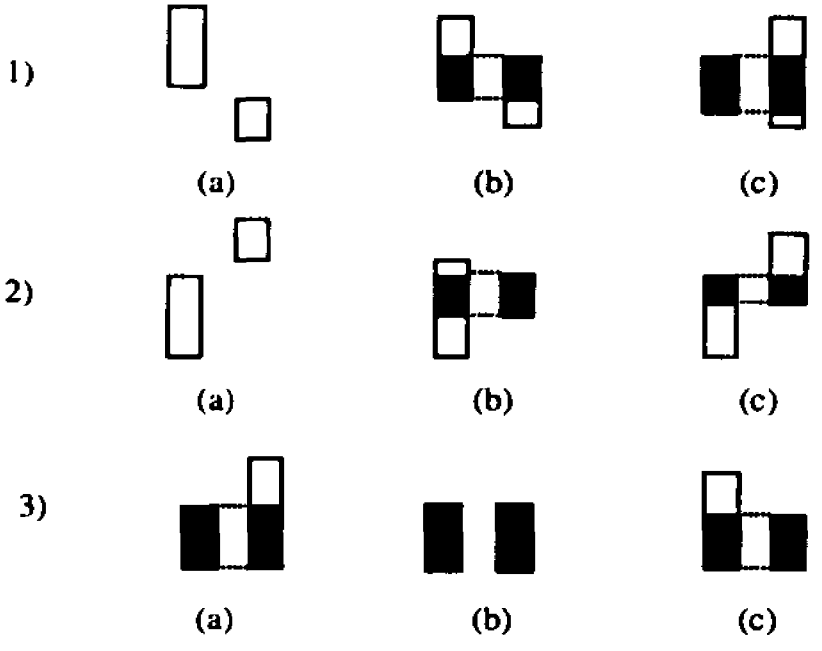
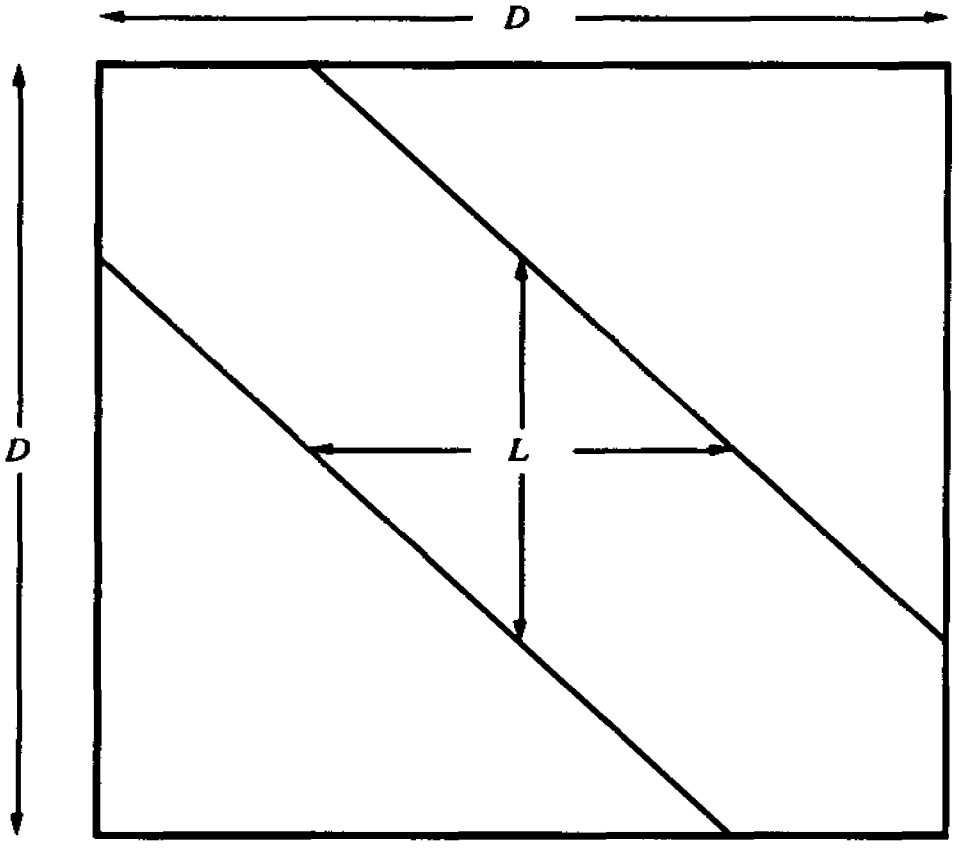


Figure 4.5 Band Matrix. A square sparse matrix of dimension D is shown. The shadowed area represents the band of nonzero elements. The symbol L is used to denote the bandwidth. It is simply the number of nonzero elements in a interior row or column of the matrix.



MATRIX_MULTIPLY Algorithm

Step 1. Use **TRANSPOSE** to obtain EB^T and SB^T;

Step 2. Let $a_1, a_2, ..., a_g$ be the row numbers of nonzero rows of A such that $a_i \leq a_{i+1}$;

Let $b_1, b_2, ..., b_h$ be the row numbers of nonzero rows of B^T such that $b_j \leq b_{j+1}$;

For i =1 **to** *g* **do**

For j=1 **to** *h* **do**

Perform vector dot-product operation to row A_{a_i} of A and row $B^T_{b_j}$ of B^T as described in the VECTOR_MULTIPLY algorithm;

Store the result C_{a_i, b_j} in EC and update SC;

endfor

endfor

end of *MATRIX_MULTIPLY*

The row numbers a_i, $1 \leq i \leq g$, and b_j, $1 \leq j \leq h$, and segment information of row a_i of A and row b_j of B^T can be extracted from EA, SA, EB^T and SB^T. Since nonzero elements and segments of A and B^T are stored in EA, SA, EB^T, SB^T in row major order, a_i, $1 \leq i \leq g$, and bj, $1 \leq j \leq h$, can be computed by linear scans of SA and SB^T, and rows A_{a_i} and $B^T_{b_j}$, which are divided into segments, can be reconstructed from EA and EB^T while scanning SA and SB^T. A detailed implementation of algorithm MATRIX_MUTIPLY is given in Appendix B.

Let r_{a_i} and s_{b_j} be the number of segments in row A_{a_i} of A and row $B^T_{b_j}$ of B^T. Then define

$$r = \sum_{1 \le i \le g} r_{a_i},$$

$$s = \sum_{1 \le j \le h} s_{b_j}, \text{ and}$$

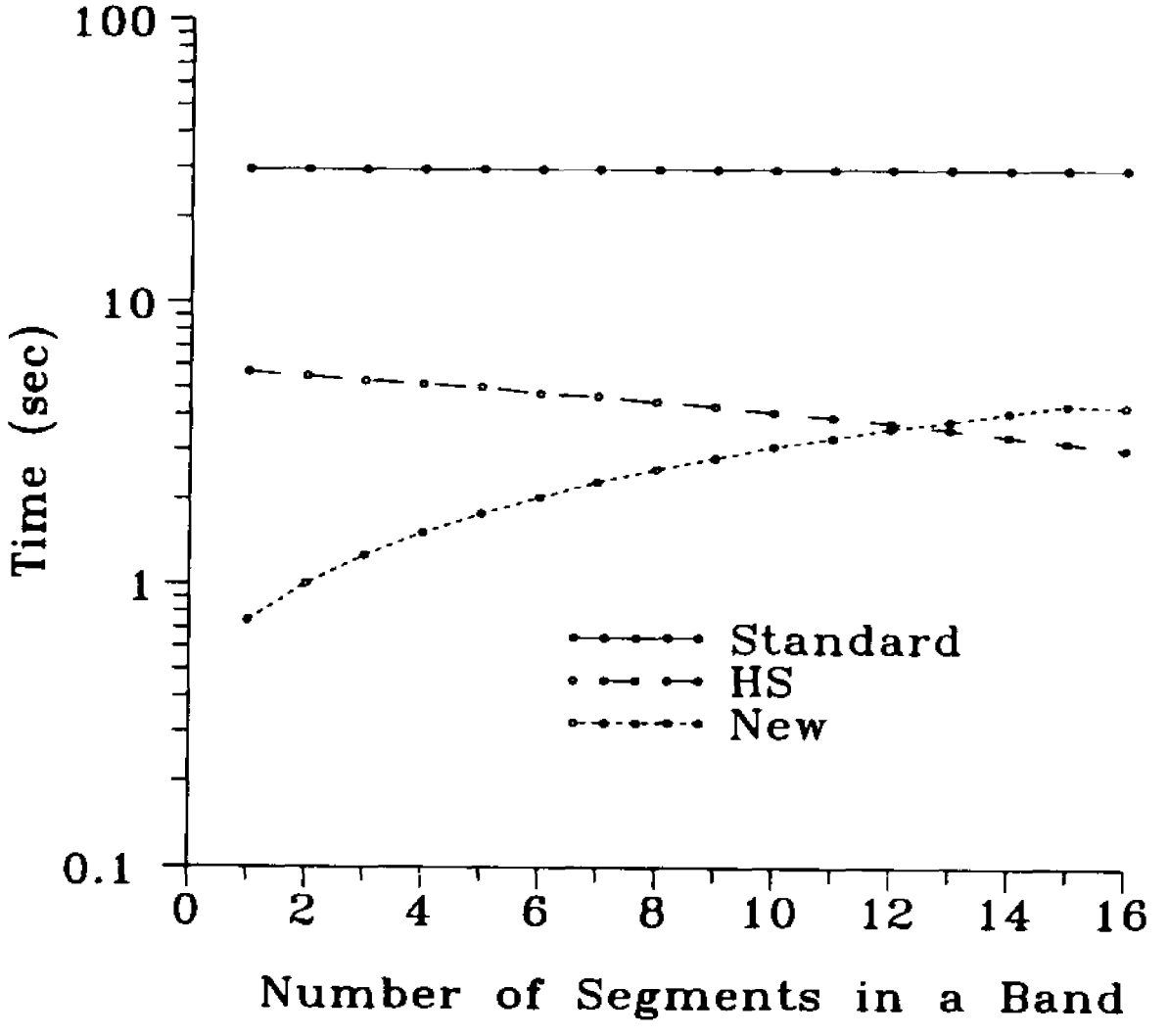
$$\text{Overlap}_{AB} = \sum_{\substack{1 \le i \le g \\ 1 \le j \le h}} \text{Overlap}_{A_{a_i} B^T_{b_j}}$$

Now since storing an element $C_{a_i b_j}$ in EC and updating SC can be done in constant time, as can be easily seen in the sub-algorithm STORE_SUM given in Appendix B, the total run time for the algorithm MATRIX_MULTIPLY is $O(h \times r + g \times s + \text{Overlap}_{AB})$. Now compare the MATRIX_MULTIPLY algorithm with the one given in [Hor 83]. The time complexity of the [Hor 83] algorithm is $O(h \times t_A + g \times t_B)$, where t_A and t_B are the number of nonzero elements in A and B, respectively. (Note that the time complexity of this algorithm was given as $O(p \times t_A + m \times t_B)$. By slightly modifying the algorithm, $O(h \times t_A + g \times t_B)$ time complexity is achievable). Since $\text{Overlap}_{AB} \le t_A + t_B$, $r \le t_A$ and $s \le t_B$, theoretically the performance of the proposed sparse matrix multiplication algorithm is no worse than that of the algorithm given in [Hor 83]. Considering the fact that, for most sparse matrix multiplication problems $r < t_A$, $s < t_B$, and $\text{Overlap}_{AB} < t_A + t_B$, the new algorithm out performs the one given in [Hor 83].

4.4 Experimental Results

The MATRIX_MUTILPLY algorithm was tested using FORTRAN on an IBM 3090/600E. The matrices A and B were taken to be identical band square matrices of dimension D as shown schematically in Figure 4.5. The size D was fixed at 300 and its bandwidth L was set at 31 with each row of both A and B initially chosen to be a single segment. Test data sets were generated from these initial distributions by iteratively and randomly breaking segments of length at least three in both A and B into two segments by replacing one of the nonzero element with 0. The performance of the algorithm, which includes TRANSPOSE and STORE_SUM as sub-algorithms, on this set of data is shown in Figure 4.6. For comparison, results for the standard and sparse matrix multiplication algorithms (see Section 4.1 and [Hor 83], respectively) using the same data are also shown. In addition to the fact that in scientific and engineering applications sparse band matrices are very common, it is also important to note that this random generation procedure yields test data with segment lengths that are normally distributed. When each row contains one segment, the results shows that the new algorithm is between 7 and 8 times faster than the algorithm given in [Hor 83] and about 40 times faster than the standard algorithm. When there are 12 segments in each row, except for the first and last 15, that is, 4 segments with one nonzero element and 8 with two elements, the algorithm performs at about the same speed as the [Hor 83] algorithm. The new algorithm is slower than the [Hor 83] algorithm when the number of segments in each row is increased beyond 12. This is because when

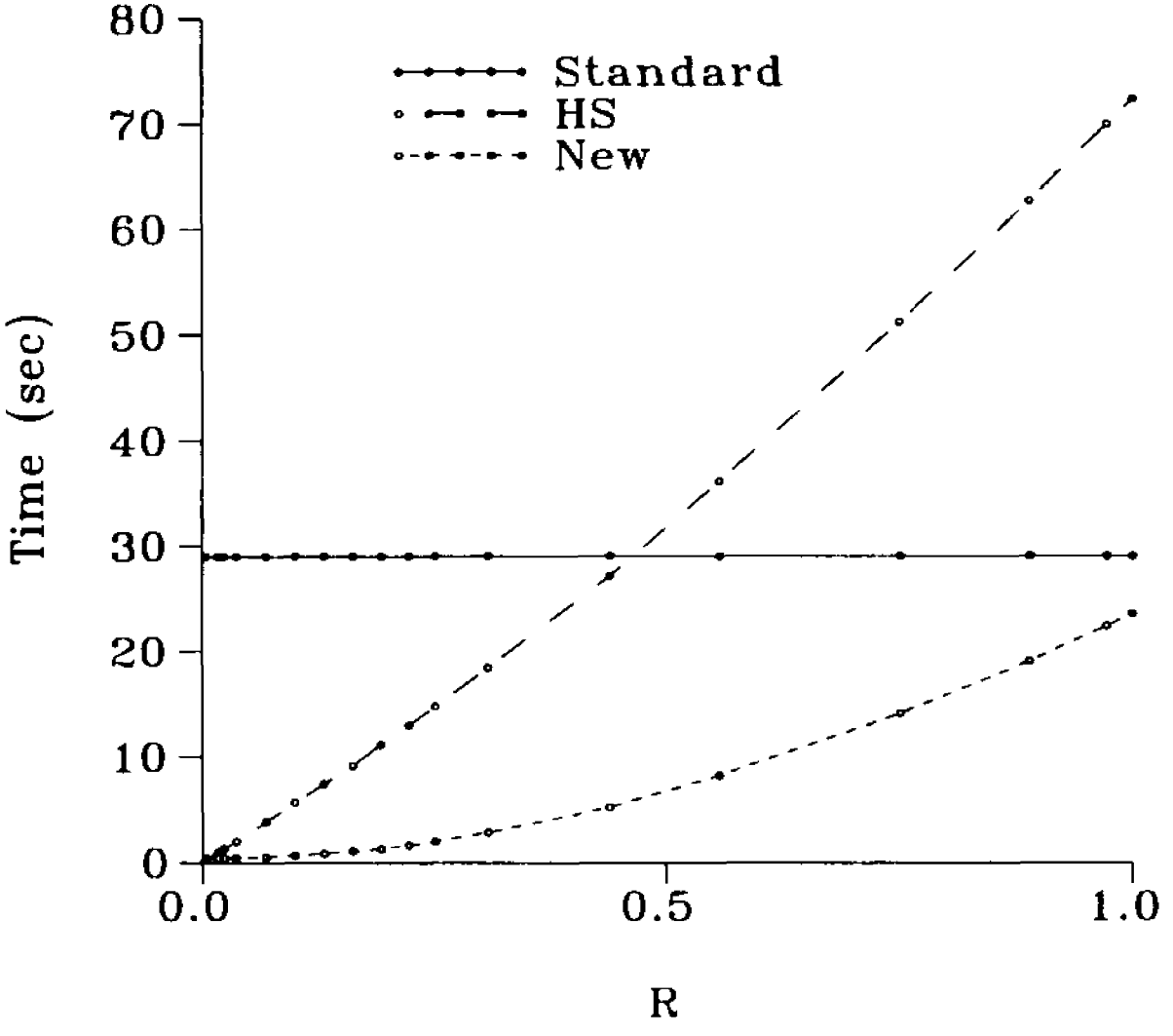
Figure 4.6 Performance test A for a sparse matrix multiplication. As an example, two 300 × 300 band matrices were multiplied using the standard (Standard), Horowitz-Sahni (HS), and new (New) matrix representations and the corresponding matrix multiplication algorithms. The value of L was fixed at 31 with one segment per row, and then these segments were broken iteratively and randomly by replacing one of the non-zero elements with 0, leaving segments of length at least three. The test was performed on an IBM 3090/600E using FORTRAN.



$r \approx t_A$ and $s \approx t_B$ the overhead in comparing segments to find overlapping intervals approaches the number of comparisons required in the [Hor 83] algorithm, a result that is made clear by a comparison of the time complexities, $O(p + r + m \times s + Overlap_{AB})$ and $O(p + t_A + m \times t_B)$, respectively.

One might expect the new algorithm to be slower than the standard matrix multiplication algorithm when the operand matrices are dense. However, experimental results can be used to demonstrate that this is not necessarily the case. For example, consider the performance of the three matrix multiplication algorithms shown in Figure 4.7 for a setup in which A and B are taken to be identical band matrices with one segment in each row. The quantity R measures the ratio of the number of nonzero elements to the total number of elements. The results show that the [Hor 83] algorithm is more than 3 times slower than the new one for the dense R = 1 case. In fact, for matrices of this type the new algorithm is *always* faster than both the [Hor 83] and standard methods. In particular, even for the dense R = 1 case, when the A and B matrices contain no zero elements, the new algorithm executes about 20% faster than the standard one. An explanation for this can be found in the way most compilers allocate space in memory for the storage of two-dimensional arrays. The elements are usually arranged in either row-major or column-major order. This means, for example, that if the two-dimensional arrays defined by the FORTRAN language used in the calculation are arranged in row-major order, during each iteration the addresses of the A_{ik} and B_{kj} factors must be calculated before the innermost $A_{ik} \times B_{kj}$ multiplication (see Section 4.1) can be performed. The overall overhead associated with the calculation of these addresses may involve as many as $O(n^3)$ hidden integer multiplications and

Figure 4.7 Performance test B for a sparse matrix multiplication. As an example, the product A × A was calculated where A was a band matrix of dimension 300 multiplications with a single one segment per row (see Figure 4.5). The length L was changed so R, the ratio of the number of nonzero elements to the total number of elements in A varied from 0 to 1. As before, this test was also performed on an IBM 3090/600E using FORTRAN.



associated addition operations, a result that can easily account for why the standard method is slower than the new one. In contrasts, while the new algorithm requires the transformation of (EB, SB) into (EB^T, SB^T), during the multiplication process only the two linear arrays EA and EB^T are scanned with an occasional reference to SA and SB^T. The array addressing in the new data structure is therefore much simpler. These results show that the conventional two-dimensional array representation may not be optimal even for operations on dense matrices.

4.5 Discussion

A new representation for sparse matrices has been introduced that can save a considerable amount of memory space, when compared with the standard 2-dimensional array representation and even the data structure introduced in [Hor 83], for many scientific and engineering applications. For example, for a 300 × 300 band matrix with L = 31 (one segment in each row), there are 9,060 nonzero elements. The new representation requires 9,963 words of memory, which is less than 36% of the 27,183 words required by the representation given in [Hor 83], and about 10.7% of the 90,000 words required for the standard 2-dimensional array representation. The associated matrix multiplication algorithm has been shown to out perform the one given in [Hor 83] for matrices containing clusters of nonzero elements, such as band or triangular matrices, that are common in many applications.

Based on comparisons of the performance of these three approaches, a general scheme for matrix multiplication can be devised. Specifically, a profile

can be associated with a matrix that gives the fraction of its elements that are nonzero and the number of segments it contains. The performance of the three matrix multiplication algorithms can then be correlated by experimentation with threshold values on the maximum and minimum, respectively, of these indicators for the matrices in the product. The preferred representation for a given problem can be selected based on average profile indicator measures for matrices that enter into the calculation and the matrices transformed accordingly into the representation that is predicted to achieve optimal performance. Since as indicated in Section 4.2, transformations between any two of the three representations can be carried out very efficiently, such an approach may lead to significant performance improvement. In Appendix B, two procedures called PACKING and UNPACKING, are given for transforming between the standard 2-dimensional matrix form and the new representation. If the elements of a matrix are input in row-major order, PACKING can be used to construct the new representation in an on-the-fly fashion while UNPACKING can be used to convert a matrix in our representation into the a standard 2-dimensional array form.

As the experimental results shown in Figures 4.6 and 4.7 indicate, a new data structure and an associated algorithm has been introduced that can yield improved matrix multiplication performance. In general, the improved performance can be expected when the factor matrices are relatively sparse matrices with the nonzero elements clustered into segment of length 1.5 or greater. However, even for relatively dense matrices the new scheme may out perform algorithms based on the standard and [Hor 83] representations for dense matrices. Beyond this, since a vector dot product is performed on the

overlap portion of segments, one from each operand matrix, in each iteration of the algorithm, it may be possible to use the vector processing to realize additional gains. In contrast, the [Hor 83] algorithm is less amenable to a vector processing adaptation.

CHAPTER 5
CONCLUSION

The goal of this dissertation research was to find more efficient ways for performing certain large-scale scientific computations. The focus was on two particular but general aspects of scientific computing, namely, reducing the repetition of intermediate computations, such as the evaluation of overlap integrals in quantum many-body physics calculations, that is common in such applications, and exploiting an efficient algorithm for performing sparse matrix operations like matrix-vector and matrix-matrix multiplications. For the first of these two, a new data structure called a weighted search tree (*wst*) was developed which supports a numerical database system such that subject to the availability of fast memory storage the redundant calculations in large-scale scientific applications can be reduced. For the second, an algorithm for the multiplication of sparse matrices, based on a new representation for sparse matrices, was developed which saves not only space but also time.

One way to reduce redundant intermediate computations in large-scale numerically intensive scientific applications is to employ a numerical database. Since most scientific applications are conducted in a nonrecursive programming environment with fixed-size array constraints, such as FORTRAN, it is highly desirable to have a data structure which will perform in such a context, and that it be minimal with respect to both the space and time computing resources it requires. The *wst* developed as part of this dissertation satisfies all of these requirements. Specifically, it is space efficient for the following four reasons:

1) the heap component of the *wst* is an implicit data structure which does not require space for left and right child pointers,
2) the AVL tree is represented as a multilist structure which needs space for only left and right links,
3) the base priority and hit frequency are combined into a single word in the priority scheme,
4) the balance factor is stored with the priority value since only two bits per node are required for it.

The *wst* is time optimal because all search, insert, and delete operations are done in $O(\log n)$ time. Thus, with the *wst*, one has available a data structure which can be implemented in any scientific application to minimize the number of redundant intermediate calculations, and thereby achieve greater efficiency. Indeed, this has already been demonstrated with the implementation of the predecessor to the *wst* in an algorithm for computing SU(3) coupling and recoupling coefficients which are needed in nuclear physics modeling applications [Aki 73, Dra 73].

A considerable reduction in computing time and memory space can often be achieved through the development of efficient algorithms. The algorithm developed in this dissertation for the multiplication of sparse matrices can lead to both time and space savings. First of all our representation of sparse matrices saves space compared to existing schemes by considering adjacent nonzero elements as matrix segments. Specifically, the representation consists of a two dimensional array which holds the segment information and a linear array where values of the nonzero elements in a matrix are stored. If the

number of segments in the matrix is less then two-third the number of nonzero elements, this new representation saves space over, for example, the Horowitz-Sahni representation. Since the sparse matrices encountered in many scientific applications are banded along the diagonal or triangular in structure, our representation can result in large space savings. For example, in a 300 × 300 band matrix with L = 31 (one segment in each row), there are 9,060 nonzero elements. Our representation requires 9,963 words, which is less than 36% of the 2,7183 words required by the Horowitz-Sahni representation [Hor 83], and about 10.7% of the 90,000 words required by a 2-dimensional array representation. Second, the algorithm we developed for the multiplication of sparse matrices using this sparse matrix representation is fast. For dense matrices and sparse matrices with nonzero elements that are distributed in clusters, such as band and triangular matrices, it is clear that the overhead for computing segment overlaps in our algorithm is a minimum. But, as shown by example, even in the general case the performance of our algorithm can be expected to be good. More importantly, since a vector dot product is performed on the overlap portion of two nonzero segments in each iteration of our algorithm, one from each of the operand matrices, it should be clear that it is possible to use vector processing hardware and procedures to further speed up the multiplication operation. Our representation is also easily obtained from and converted into other representations, in particular the standard two-dimensional array and Horowitz-Sahni representations. This is illustrated by the packing and unpacking subroutines that are given in Appendix B for this purpose. To reiterate, our sparse matrix algorithm is fast, space saving, and easily converted to two other standard representations.

The two techniques presented here were motivated by and have been used to solve a problem in nuclear physics. Specifically, a research area which is currently very active within the nuclear physics community is that describing the collective rotational behavior of nuclei, i.e. motion involving the rotation of the nucleus as a whole. It has been determined through various investigations [Cas 89] that the residual interactions which are important for obtaining a good description of the observed rotational features in nuclei are those contained in the rotationally invariant many-body hamiltonian

$$H = H' + \sum_{n=1}^{A} (C_n \cdot s_n + D_n l_n^2) + (AJ^2 + BK_j^2) + P$$

$$H' = H_0 - \frac{1}{2}\chi[Q_2 \cdot Q_2 - (Q_2 \cdot Q_2)^{TE}] . \qquad (5.1)$$

The first term H_0 is the mean field harmonic oscillator hamiltonian, while the remainder are the residual interactions needed to account for deformation, single-particle energy splitting, rotational energy splitting, and nucleon pairing, respectively. The interactions appear in (5.1) in the order of their importance with regard to influencing collective rotational features in nuclei. Given this many-body hamiltonian, describing rotational motion now requires constructing its matrix representation for states of a fixed total angular momentum J, and diagonalizing the matrix to obtain the energy eigenvalues and eigenstates that contain the information on the state of the system. These matrix representations for the hamiltonian of (5.1) are very large and also rather sparse, which is due to the symmetry basis employed being optimal with regard to describing collective rotational motion. For example a typical

hamiltonian matrix is found to be only approximately 10% dense. This significantly reduces the amount of computer resources which are required for constructing and diagonalizing the matrices. However, the very large dimensions of these sparse matrices still result in considerable demands being made upon computer resources. If such calculations are to become feasible, then it is clear that computational techniques must be devised which more rapidly and efficiently solve this large sparse eigenmatrix problem. The two techniques presented above help accomplish this goal.

Since in the matrix construction of the hamiltonian (5.1) for any given angular momentum J the same coupling coefficients and reduced matrix elements reoccur frequently, it is desirable to save calculated intermediate results and thereby avoid their wasteful regeneration by using databases built on the *wst* structure. The specific coupling coefficients and reduced matrix elements needed in the construction of the sparse matrices of the hamiltonian (5.1) are calculated only once, while unneeded values are never generated. Moreover, under a fixed memory size constraint, those values which are the most expensive to generate are retained in favor of the least expensive. This means that the required matrix representations can be generated using a minimum amount of CPU time, as the nonzero matrix elements are computed only once and in the most efficient manner possible. As an example, the implementation of the *wst* data structure has resulted in typical gains in run times of at least a factor of two for the case of ^{20}Ne [Roc 91]. This kind of reduction in CPU time utilization is indicative of the savings that can be expected when a *wst* numerical database is implemented.

Once the sparse matrix representations of the hamiltonian given by (5.1) have been constructed, one is confronted with the task of diagonalizing them in as efficient a manner as possible in order to obtain the energy eigenvalues and eigenvector wavefunctions. The best approach is clearly to exploit their sparse nature and employ a method where the memory storage required for the matrices is minimal and the calculations involving the zero matrix elements are avoided. All standard diagonalization algorithms for real symmetric matrices of a general form, be they either of modest or large size, reduce the full matrix to tridiagonal form as an intermediate step. When all eigenvalues and eigenvectors are desired the reduction from full to tridiagonal form is accomplished by a series of similarity transformations using reflector matrices, which is very costly when dealing with very large matrices. When only a few of the extremal eigenvalues and their eigenvectors are desired, as in the present case, this reduction is achieved at much less cost by using the Lanczos method which constructs a guess of the tridiagonal matrix given a trial starting vector [Cul 85]. Once the tridiagonal form is achieved the QL and QR algorithms [Gol 89, Pres 89] with quotient shifts and plane Jacobi rotations are employed to complete the diagonalization process. In both cases, matrix-matrix or matrix-vector multiplications are performed that involve the original matrix and its descendants that arise during execution of the algorithm. If the original matrix is sparse, then in both cases it is reasonable to expect that the successive operations will involve matrices which are sparse. A means of performing these calculations more rapidly is to therefore devise and implement a more efficient algorithm for sparse matrix-matrix and matrix-vector multiplications.

In conclusion, a numerical database system constructed by using a weighted search tree was developed. This allows one, for example, to minimize the redundant intermediate calculations that occur in constructing the matrix representation of the many-body hamiltonian when limited by fixed memory space constraints. This numerical database system is simple to use and because the coded algorithm is generic and passive it can be implemented in many scientific applications as well as for many different data structures within a single application to minimize the use of computer resources. For example, it has been found to yield gains of at least a factor of two in typical nuclear physics shell-model applications. With regard to the diagonalization of large sparse matrices, a new matrix representation has been developed, along with a matrix multiplication algorithm, which holds the potential to significantly diminish the cost of storing and reducing large sparse band matrices to tridiagonal form (an important intermediate step in all diagonalization processes). With the sparse matrix multiplication algorithm and the *wst* developed as a part of this research project, the cost of performing the quantum physics computations required for the many-body description of collective nuclear rotations has been significantly reduced, and therefore the scope of applications that can be undertaken broadened. Our hope is that other researchers working in the same as well as other areas will likewise find the results useful.

Considering remarkable advances in high performance parallel computer architectures made in recent years, we believe that an important future study will be to generalize the concepts of our *wst* and new representation of a sparse matrix to the parallel processing environment. For example, our data

structure and *wst* algorithms could be developed into efficient parallel programs [Hwa 84, Qui 87] for manipulating heaps and AVL trees. One approach might be to design a special purpose architecture for numerical databases that can be attached to a general purpose supercomputer so that the performance of the system on large-scale applications can be significantly improved. For example, a straightforward hardware implementation of our numerical database would be a content-addressable (associative) memory which would automatically update the user defined priority of data items after every memory access, and replace a data item by a new one when necessary. Regarding our new sparse matrix representation and the associated algorithms we note that the overlap segment concept could be used to develop new (parallel) algorithms for other matrix operations [Chr 89, Ewe 90, Kim 89, Len 89, Qui 87, Sim 89]. Furthermore, it may be possible to introduce hardware components based on the segment-algorithm concept into future generation computers to achieve additional performance improvements in large-scale scientific and engineering applications.

BIBLIOGRAPHY

[Ade 62] G. M. Adel'son-Vel'skii and E. M. Landis, "An algorithm for the organization of information," Sov. Math. Dokl. **3**, 1250-1262, 1962.

[Aho 74] A. V. Aho, J. E. Hopcroft and J. D. Ullman, "*The Design and Analysis of Computer Algorithms*," Addison-Wesley, Reading, MA, 1974.

[Aki 73] Y. Akiyama and J. P. Draayer, "A User's Guide to Fortran Programs for Wigner and Racah Coefficients of SU(3)," Comp. Phys. Comm., **5**, 405-415, 1973.

[Ben 75] J.I Bentley, "Multidimensional Binary Search Trees Used for Associative Searching", *CACM*, **18**, 9, 1975.

[Bur 85] R. L. Burdin and J. D. Faires, "*Numerical Analysis*," Third Edition, PWS, Boston, MA, 1985.

[Cha 84] H. Chang and S. Sitharama Iyengar, "Efficient Algorithms to Globally Balance a Binary Search Tree," Communications of the ACM, **27**, No. 7, July 1984

[Che 84] C.-C. Chen, P. Hernon and etc.,"*Numeric Databases*," Ablex Pub. Co., Norwood, NJ, 1984

[Chr 89] A. T. Chronopoulos and C. W. Gear "S- step Iterative Methods for Symmetric Linear Systems," J. Comp. App. Math., **25**, 153-168, 1989

[Cul 85] J. K. Cullum and R. A. Willoughby, "*Lanczos Algorithms for Large Symmetric Eigenvalue Computations*," Vol. 1, Birkhauser Boston, Germany, 1985

[Dat 86] C. J. Date, "*An Introduction to Database Systems*," Fourth Ed., Addison-Wesley, Reading, MA, 1986.

[Dra 73] J. P. Draayer and Y. Akiyama, "Wigner and Racah Coefficients for SU(3)," J. Math. Phys., **14**, 1904-1912, 1973

[Ewe 90] L. Magnus Ewerbring and Franklin T. Luk, "Computing the Singular Value Decomposition on the Connection Machine," IEEE Transactions on Computers, **39**, No. 1, 152, 1990.

[Fos73] C. C. Foster, "A Generalization of AVL Trees," *Comm. ACM*, **16**, No. 8, 513-517, 1973

[Gol 89] G. H. Golub and C. F. Van Loan, "*Matrix Computations*," Second Ed., The Johns Hopkins University Press, Baltimore, MD, 1989.

[Hel 86] P. Helman and R. Veroff, "*Intermediate Problem Solving and Data Structure*," Benjamin.Cummings Pub., Monlo Park, CA, 1986.

[Hwa 84] K. Hwang and F. A. Briggs, "*Computer Architecture and Parallel Processing*," Mcgraw-Hill, NY, 1984.

[Hor 83] E. Horowitz and S. Sahni, "*Fundamentals of Data Structures*," Computer Science Press, Rockville, MD, 1983.

[IBM 87] IBM, "*VS FORTRAN Version 2: Language and Library Reference*," Release 3 IBM, San Jose, CA, 1987.

[Iye 85] S. Sitharama Iyengar and Hsi Chang, "Efficient Algorithms to Create and Maintain Balanced and Threaded Binary Search Trees," Software-Practice and Experience, 15, 925-941, Oct. 1988

[Kim 89] S. Kim and A. T. Chronopoulos, "S-step Lanczos and Arnoldi Methods on Parallel Computers," Tech. Rep. TR89-83, Computer Science Department, University of Minnesota, 1989.

[Knu 73a] D. E. Knuth, "*The Art of Computer Programming*," Vol. 1, Addison-Wesley, Reading, Ma, 1973

[Knu 73b] D. E. Knuth, "*The Art of Computer Programming*," Vol. 3, Addison-Wesley, Reading, Ma, 1973

[Kor 86] H. F. Korth and A. Silberschatz, "*Database System Concepts*," McGraw-Hill, NY, 1986

[Kro 79] L. I. Kronsjo, "*Algorithms: Their Complexity and Efficiency*," John Wiley and Sons, NY, 1979

[Len 89] S. Lennart Johnsson, Tim Harris and Kail K. Mathur, "Matrix Multiplication on the Connection Machine," Proceedings of Supercomputing '89, 326, 1989

[Lip 86] S. Lipschutz, "*Schau's Outline of Theory and Problems of Data Structures*," McGraw-Hill, New York, NY, 1986.

[Mac 83] B. J. MacLennan, "*Principles of Programming Languages: Design, Evaluation, and Implementation*," Holt, Rinehart and Winston, New York, NY, 1983.

[Neg 88] J. W. Negele and H. Orland, "*Quantum Many-Particle Systems*," Addison-Wesley, Reading, MA, 1988.

[Net 89] J. Neter, W. Wasserman and M. H. Kutner, "*Applied Linear Regression Models*," Second, Irwin, Homewood, IL, 1989.

[Par 89a] S. C. Park and J. P. Draayer, "Balanced Binary Tree Code for Scientific Applications," *Comp. Phys. Comm.*, **55**, 189, 1989.

[Par 90b] S. C. Park, J. P. Draayer, and S. -Q. Zheng, "Priority Balanced Binary Tree for Large-Scale Scientific Applications," Technical Report #90-015, Department of Computer Science, Louisiana State University, 1990.

[Par 90c] S. C. Park, J. P. Draayer, and S.-Q. Zheng, "Time-Space Optimal Numerical Database for Large-Scale Scientific Applications," in Proceedings of the International Computer Symposium, Hsinchu, Taiwan, R.O.C, December 17-19, **333**, 1990.

[Par 91d] S. C. Park, J. P. Draayer, and S.-Q. Zheng, "An Efficient Algorithm for Sparse Matrix computations," Technical Report #91-009, Department of Computer Science, Louisiana State University, 1991.

[Parl 80] B. N. Parlett, "*The Symmetric Eigenvalue Problem*," Prentice-Hall, Englewood Cliffs, NJ, 1980.

[Pet 85] J. L. Peterson and A. Silberschatz, "*Operating System Concepts*," Addison-Wesley Pub. Co., Reading, MA, 1985.

[Pre 85] F.P. Preparata and M.I. Shamos, "*Computational Geometry*," Springer-Verlag, 1985.

[Pres 89] W. H. Press, B. P. Flannery, S. A. Teukolsky and W. T. Vetterling, "*Numerical Recipes; The Art of Scientific Computing*

(Fortran Version)," Cambridge University Press, New York, NY, 1989.

[Qui 87] M. J. Quinn, "*Designing Efficient Algorithms for Parallel Computers*," McGraw-Hill, NY, 1987.

[Rei 83] E. M. Reingold and W. J. Hansen, "*Data Structures*," LBC, Boston, MA, 1984.

[Roc 91] P. Rochford, S. C. Park, J.P. Draayer and S.-Q. Zheng, "Optimal Methods For Large_Scale Scientific Database and Sparse Matrix Applications," to be appeared in the AIP Proceedings of the Conference on Computational Quantum Physics held May 22-25, 1991, at Vanderbilt University, Nashville, TN.

[Sim 89] H. D. Simon, "*Scientific Applications of the Connection Machine*," World Scientific, NJ, 174, 1989.

[Stra 86] V. Strassen, "The Asymptotic Spectrum of Tensors and the Exponent of Matrix Mulitplication," IEEE, **49**, 1986.

[Ull 89] J. D. Ullman, "*Principles of Database and Knowledge-based Systems*," Computer Science Press, Rockville, MD, 1989.

[Wir 76] N. Wirth, "*Algorithms + Data Structures = Programs*," Prentice-Hall, Englewood Cliffs, NJ, 1976.

[Zhe 89] S.-Q. Zheng, "A Simple and Powerful Representation of Binary Search Trees", Technical Report #89-016, Department of Computer Science, Louisiana State University, 1989.

APPENDIX A

CODES FOR A NUMERICAL DATABASE SYSTEM

A.1 Program Summary

Tile of program: WSTREE

Computer: IBM 3090/600E

Operating system: MVS/XA (Version 3, Release 13)

Programming language used: FORTRAN

High speed storage required: The six routines that comprise the weighted search tree (*wst*) require a total of 20724 bytes of memory space on the IBM 3090 mainframe computer. The main program and the tree and data buffer storage requirements are over and above this base amount.

Peripherals used: none

No. of lines in program: 1929 (319 in sample DRIVER, 1610 in WSTREE package)

Keywords: heap, binary tree, AVL tree, weighted search tree, linked list, multilist representation, dynamic storage, priority strategy, numerical database, on-line database, file directory system

Nature of the physical problem

Scientific computing applications frequently involve redundant calculations. This occurs either because the number of numbers that need to be calculated is too large to be stored in memory or they occur in unknown combinations so pregeneration, which would allow them to be ordered separately and efficiently so a simple binary look up could be used, is impracticable or impossible. The *wst* numerical database system introduced here [1-3] can be used to circumvent this problem as it enables one to perform the on-line search of an existing data structure for a particular element and use it if it is found, but if it is not found, it allows generated results to be added to the list so they can be reused as necessary at a later stage in the calculation. Typically this *modus operandi* yields a two fold gain, namely, needed results are calculated only once and unneeded results are never generated.

When the database is full, which occurs when either the tree or associated storage arrays have reached their maximum capacity, one or perhaps even several elements, depending on the size of the incoming data set, have to be deleted before new information can be added to the database. In the sample program the

delete option is exercised when the incoming element has a higher priority than the lowest priority element in the database. The delete process checks to see if the free space generated by deleting the lowest priority node suffices to accommodate the incoming data, and if it does not additional nodes are deleted, provided of course that their priorities are less than that of the incoming node, until sufficient free space is obtained. Whenever the delete option is invoked, it is the lowest priority node that is removed from the database.

Method of solution

A *wst* system solves the problem with a series of routines that have optimal time and space complexities. Within the *wst*, search, insert and delete operations on dynamically generated lists of length n execute in times that goes as $O[\log(n)]$. In addition, the *wst* saves space by using a multilist representation for the height balanced tree that is the backbone structure of the database system. A full discussion of the theory behind the *wst* structure can be found in Ref. [3].

Restrictions on the complexity of the problem

The maximum number of nodes in a *wst* is set by user. If the pointers are n bit integer words, the theoretical limit on the maximum number is 2^n-1. In actual practice, however, the number will usually be much less than this.

Typical running time: 0.14 ms/item on an IBM 3090/600E

Unusual features of the program

Intrinsic logical AND and SHIFT functions for integers, which are called IAND and ISHFT in the *wst* routines [4], are used for bit operations to encode and decode the priority and balance factor that are stored as a single integer word.

References

[1] C. J. Date, "*An Introduction to Database Systems*," Fourth Ed. (Addison-Wesley, Reading, MA, 1986).

[2] S. C. Park and J. P. Draayer, *Comput. Phys. Commun.* **55**, 189, 1989.

[3] S. C. Park, J. P. Draayer, and S.-Q. Zheng, "Time-Space Optimal Numerical Database for Large-Scale Scientific Applications," in Proceedings of the International Computer Symposium, December 17-19, 1990, Hsinchu, Taiwan, R.O.C.

[4] IBM, "*VS FORTRAN Version 2: Language and Library Reference*," Release 3 IBM, San Jose, CA, 1987.

A.2 Sample Application Program Using the Weighted Search Package

```
C ------------------------------------------------------------------
C
C             ********************************************************
C             ****          SAMPLE APPLICATION PROGRAM            ****
C             ****                    using the                   ****
C             ****    WEIGHTED SEARCH TREE (WSTREE) PACKAGE       ****
C             ********************************************************
C
C ------------------------------------------------------------------
C
C Authors:  Soon Park, J. P. Draayer and S.-Q.Zheng
C           Departments of Computer Science/Physics and Astronomy
C           Louisiana State University
C           Baton Rouge LA
C           USA 70803-4001
C
C                   BITNET:   PHDRYR @ LSUMVS or LSUVM
C                   TELEX:    559184
C                   PHONE:    USA-504-388-2261
C                   FAX:      USA-504-388-5855
C
C Version: 1.1      LSU (07/01/91)
C
C ------------------------------------------------------------------
C
C Updates: 07/90 Original from a FORTRAN code written by Soon Park
C
C ------------------------------------------------------------------
C
C The sample application program has three parts:
C
C   1.  DRIVER -->  A main program illustrating the use of routines
C                   from the WSTREE package.  Sample fetch, insert and
C                   delete operations are performed on a tree with the
C                   keys from an integer random number generator. The
C                   program uses the system functions STKLOK (start
C                   clock) and KLOK (clock) for c.p.u. time measures.
C   2.  IRAND   -->  Integer random number generating function for keys,
C                   data, and priorities used in the sample program.
C   3.  ISIZE   -->  Integer random number generating function for the
C                   data size used in the sample program.
C
C This accompanying "WST" package consists of 6 different subroutines:
C
C   1. TSET     -->  initializes a storage area for use as a binary tree
C        TSETFB -->  ... entry in TSET for a fixed-buffer size structure
C        TSETMB -->  ... entry in TSET for a mixed-buffer size structure
C   2. TCHK     -->  performs the search operation on the data structure
C   3. TADD     -->  add a new element to an existing WST data structure
C        TADDFB -->  ... entry in TADD for addition of fixed-buffer item
C        TADDMB -->  ... entry in TADD for addition of mixed-buffer item
C   4. TINS     -->  called by TADD to insert new node into existing WST
C   5. TDEL     -->  called by TADD to delete low priority node in a WST
```

```
C   6. TOUT     -->  generates output information on specific tree nodes
C
C When the data associated with a node in the WST is variable length
C the link-list option must be used. Three things must change in going
C from a FB (... fixed-buffer) to a MB (... mixed-buffer) application:
C
C   1. Dimension of LLBUFF, namely, LLBUFF(-2:0) --> LLBUFF(-2:MXBUFF)
C   2. TSETMB rather than TSETFB must be called for the initialization
C   3. TADDMB rather than TADDFB must be called to add database entries
C
C ---------------------------------------------------------------------
C
C                              ***************
C                              *** DRIVER ***
C                              ***************
C
C This is a sample program designed to test routines in the weighted
C search tree (WST) package. The key, data and priority (weight) are
C generated by calls to integer random number generators. The driver
C program builds a WST with keys that are five integer word in length.
C The location and size of the data are also determined randomly. The
C maximum number of items in the sample program is set at 25000, hence
C the dimension ID(-10:250010). The first 11 elements, ID(-10:0), give
C the tree parameters and the next 250010 are for the node information:
C 5 keys, location of data, size of data, balance factor and priority,
C left-link pointer, and right-link pointer for each. The last 10 are
C for bookkeeping information and are used as a temporary node with the
C same structure as regular nodes. For details on how this is done, see
C the documentation in subroutines TSET, TCHK, TINS and TDEL. The buffe
C array size is set at 50000 real numbers and individual data sets with
C size between 1 to 5 are generated by a random number generator called
C ISIZE. The free space and each set of data are managed by a pointer
C in LLBUFF, using a linked list allocation method. The logic followed
C by the main program is the following:
C
C Step 0) generate key of new item to be inserted into the dataset
C Step 1) check if a node with the same key is already in the tree
C         if yes, retrieve the location and size and goto Step 4)
C         else    goto step 2)
C Step 2) check if either the tree or buffer is full
C         if yes, check if the priority of incoming item is higher
C                 than that of the lowest priority node in the tree
C                 if yes, delete lowest priority node and goto Step 2)
C                 else    goto step 4)
C         else    goto Step 3)
C Step 3) insert the incoming item into the database
C Step 4) goto step 0) until 50000 iterations
C Step 5) print output
C
C ---------------------------------------------------------------------
C
C Main Program
C
C
C ////////////////////////// REQUIRED BLOCK 1 \\\\\\\\\\\\\\\\\\\\\\\\
```

```
C
C Specify WSTREE information:
C
C     PARAMETER (NKEY=5,NDAT=1,MXNODE=25000)      ! *** fixed-buffer case
      PARAMETER (NKEY=5,NDAT=2,MXNODE=25000)      ! *** mixed-buffer case
      PARAMETER (MXSIZE=(MXNODE+1)*(NKEY+NDAT+3))
      INTEGER   NEWKEY(NKEY),NEWDAT(NDAT),ID(-10:MXSIZE)
C
C         NKEY = Number of integer words per node dedicated to the key
C         NDAT = Number of integer words per node dedicated to the data
C                (NDAT.GE.1 for a fixed-buffer size application, TADDFB
C                ... store additional integer data in NEWDAT(2:NDAT),
C                    NEWDAT(1:2) is reserved for pointer and counter
C                (NDAT.GE.2 for a mixed-buffer size application, TADDMB
C                ... store additional integer data in NEWDAT(3:NDAT),
C                    NEWDAT(1:1) is reserved for pointer information
C       MXNODE = Maximum number of nodes the WST array will accommodate
C       MXSIZE = Calculated size of the WST array, given NKEY and NDAT
C
C
C Specify BUFFER information:
C
      PARAMETER (MXBUFF=50000,MXLOAD=5)
C     INTEGER   LLBUFF(-2:0)                    ! *** fixed-buffer option
      INTEGER   LLBUFF(-2:MXBUFF)               ! *** mixed-buffer option
C     INTEGER   INLOAD(MXLOAD),INTGER(MXBUFF)   ! *** check TADD type too
      REAL      BULOAD(MXLOAD),BUFFER(MXBUFF)   ! *** check TADD type too
C     REAL*8    BULOAD(MXLOAD),BUFFER(MXBUFF)   ! *** check TADD type too
C
C       MXBUFF = Size of the buffer (BUFFER) where information is store
C       MXLOAD = Size of the work area (BULOAD) for incoming informatio
C
C Work arrays used for data:
C
C       NEWKEY = Integer array holding the new (incoming) key
C       NEWDAT = Integer array holding the new (incoming) data
C       BULOAD = Real array holding the new (incoming) real data
C
C \\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\ COMPLETE ///////////////////////////////
C
C Initialize the database variables using input parameters and arrays
C
C /////////////////////////// REQUIRED BLOCK 2 \\\\\\\\\\\\\\\\\\\\\\\\\
C
C     CALL TSETFB(ID,MXNODE,NKEY,NDAT,LLBUFF,MXBUFF)
      CALL TSETMB(ID,MXNODE,NKEY,NDAT,LLBUFF,MXBUFF)
C
C \\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\ COMPLETE ///////////////////////////////
C
C Select the seed for random number generation
C
      ISEED=13
C
C Start the clock
C
```

```
      CALL STKLOK
C
C Set the number of items higher than the maximum for test purposes
C
      NOITEM = 2*MXNODE
C
      DO 500 I=1,NOITEM
C
C Generate a key
C
      DO 100 J=1,NKEY
100      NEWKEY(J)=IRAND(ISEED)
C
C Call TCHK and branch as follows:
C        To 300 if search indicates key exists
C        Simply continue under a normal return
C
C /////////////////////////// REQUIRED BLOCK 3 \\\\\\\\\\\\\\\\\\\\\\\\\
C
      CALL TCHK(NEWKEY,ID,*300)
C
C \\\\\\\\\\\\\\\\\\\\\\\\\\\\\ COMPLETE /////////////////////////////
C
C ** Normal Return (TCHK) **
C
C    Item not in the database so it must be calculated
C
C ...generate the number of real data elements using ISIZE
C
C     NOSIZE=3                                          ! *** fixed-buffer case
      NOSIZE=ISIZE(ISEED)                               ! *** mixed-buffer case
C
C ...generate the new real data (junk for sample program)
C
      DO 200 J=1,NOSIZE
200      BULOAD(J)=FLOAT(J)
C
C ...generate additional integer data for WST here ... none
C
C ...generate a base priority for node (1 to 255 using IRAND)
C
      NPBASE=IRAND(ISEED)
C
C ...add the element just generated to the existing database
C
C /////////////////////////// REQUIRED BLOCK 4 \\\\\\\\\\\\\\\\\\\\\\\\\
C
C     CALL TADDFB(NEWKEY,NEWDAT,BULOAD,NOSIZE,NPBASE,ID,BUFFER,LLBUFF)
      CALL TADDMB(NEWKEY,NEWDAT,BULOAD,NOSIZE,NPBASE,ID,BUFFER,LLBUFF)
C
C \\\\\\\\\\\\\\\\\\\\\\\\\\\\\ COMPLETE /////////////////////////////
C
      GO TO 500
C
C ** Return 300 (TCKH) **
```

```
C
C    Identical key in the tree so fetch and print node information
C
C ////////////////////////// REQUIRED BLOCK 5 \\\\\\\\\\\\\\\\\\\\\\\\\
C
C ...setup the position of the current node
C
300   ISKIP=ID(-5)
C
C ...pull the total number of data elements
C
C     NOSIZE=3                                        ! *** fixed-buffer case
      NOSIZE=ID(ISKIP+NKEY+2)                         ! *** mixed-buffer case
C
C ...extract the data from the buffer array
C
      IPOS=ID(ISKIP+NKEY+1)
      DO 400 J=1,NOSIZE
         BULOAD(J)=BUFFER(IPOS)
C        IPOS=IPOS+1                                  ! *** fixed-buffer case
         IPOS=LLBUFF(IPOS)                            ! *** mixed-buffer case
400   CONTINUE
C
C \\\\\\\\\\\\\\\\\\\\\\\\\\\\\\ COMPLETE //////////////////////////////
C
C ...print the node pointer, key, and data
C
      WRITE(6,1000) ISKIP,(ID(ISKIP+J),J=1,ID(-3))
500   CONTINUE
C
C Print the time
C
      CALL KLOK(J)
      TIME=0.01*J
      AVE=TIME/NOITEM
      WRITE(6,*)
      WRITE(6,*) 'C.P.U. TIME ==>',TIME,'   AVE. TIME ==>',AVE
C
      WRITE(6,*)
      WRITE(6,*) ' *** PROGRAM RAN SUCCESSIVELY! ***'
      WRITE(6,*)
      WRITE(6,*) '     *************************'
      WRITE(6,*) '     **   OUTPUT OPTIONS    **'
      WRITE(6,*) '     *************************'
600   WRITE(6,*)
      WRITE(6,*) ' VIEW THE NODES?  ...SELECT AN OPTION:'
      WRITE(6,*)
      WRITE(6,*) '   1 = ASCENDING ORDER'
      WRITE(6,*) '   2 = DESCENDING ORDER'
      WRITE(6,*) '   3 = QUIT OR STOP LOOKING'
      WRITE(6,*) '   ?'
C
      READ(5,*,END=700) NAD
      WRITE(6,1100) NAD
      IF (NAD.GE.3) GO TO 700
```

```
      WRITE(6,*) 'ENTER THE START POSITION'
      READ(5,*,END=700) NUM1
      WRITE(6,1100) NUM1
      WRITE(6,*) 'ENTER THE FINAL POSITION'
      READ(5,*,END=700) NUM2
      WRITE(6,1100) NUM2
      WRITE(6,*) 'WHAT STEP SIZE DO YOU WANT?'
      READ(5,*,END=700) NSTEP
      WRITE(6,1100) NSTEP
      WRITE(6,*) 'WHERE DO YOU WANT TO SEND THE DATA?'
      READ(5,*,END=700) NFILE
      WRITE(6,1100) NFILE
      CALL TOUT(NFILE,NAD,NUM1,NUM2,NSTEP,ID)
      GOTO 600
700   CONTINUE
      STOP
1000  FORMAT(1X,'EXISTS:',2X,I10,12I5/(20X,12I5))
1100  FORMAT('0 ***** SELECTED VALUE:',I5,2X,'<=====')
      END
C -----------------------------------------------------------------------
C
C                          **********************
C                          *** IRAND & ISIZE ***
C                          **********************
C
C The function IRAND generates a random integer number in the interval
C from 1 to 255. The parameter ISEED is a user supplied seed number.
C
C The function ISIZE generates a random integer number in the interval
C from 1 to 5. The parameter ISEED is a user supplied seed number.
C
C -----------------------------------------------------------------------
C
      FUNCTION IRAND(ISEED)
C
      ISEED=MOD(25173*ISEED+13849,65536)
      IRAND=INT(255.*ISEED/65536)+1
      RETURN
      END
C
      FUNCTION ISIZE(ISEED)
C
      ISEED=MOD(25173*ISEED+13849,65536)
      ISIZE=INT(5.*ISEED/65536)+1
      RETURN
      END
C -----------------------------------------------------------------------
C
C                        ****************************
C                        ***   WSTREE PACKAGE     ***
C                        ****************************
C
C                  ------------------------------------
C
C              ***********************************************
```

```
C              ***     WEIGHTED SEARCH TREE ROUTINES     ***
C              ***                  for                  ***
C              ***  SCIENTIFIC (FORTRAN) APPLICATIONS    ***
C              ********************************************
C
C ---------------------------------------------------------------------
C
C Authors: Soon Park, J. P. Draayer and S.-Q. Zhang
C          Departments of Computer Science/Physics and Astronomy
C          Louisiana State University
C          Baton Rouge LA
C          USA 70803-4001
C
C                  BITNET:  PHDRYR @ LSUMVS or LSUVM
C                  TELEX:   559184
C                  PHONE:   USA-504-388-2261
C                  FAX:     USA-504-388-5855
C
C Version: 1.1     LSU (07/01/91)
C
C ---------------------------------------------------------------------
C
C Updates: 07/90 Original from a FORTRAN code written by Soon Park
C
C ---------------------------------------------------------------------
C
C General comments on the package:
C
C   This package is written in FORTRAN since most scientific programs
C   require FORTRAN compatibility and for it the existing scientific
C   subroutine libraries are the most extensive and efficient.
C
C   The tree is a linear array ID(-10:*) consisting of eleven integers,
C   ID(-10:0), that specifies the structure of the tree, see TSET for
C   documentation on this, and node information starting with ID(1:1).
C   The latter includes the key(s), data, balance factor and priority,
C   as well as the left and right child pointers. See the documentation
C   on each subroutine for further details.
C
C ---------------------------------------------------------------------
C
C This numerical database package consists of 6 different subroutines:
C
C   1. TSET     -->  initializes a storage area for use as a binary tree
C       TSETFB  -->  ... entry in TSET for a fixed-buffer size structure
C       TSETMB  -->  ... entry in TSET for a mixed-buffer size structure
C   2. TCHK     -->  performs the search operation on the data structure
C   3. TADD     -->  add a new element to an existing WST data structure
C       TADDFB  -->  ... entry in TADD for addition of fixed-buffer item
C       TADDMB  -->  ... entry in TADD for addition of mixed-buffer item
C   4. TINS     -->  called by TADD to insert new node into existing WST
C   5. TDEL     -->  called by TADD to delete low priority node in a WST
C   6. TOUT     -->  generates output information on specific tree nodes
C
C The four routines TSET, TCHK, TADD and TOUT subject to user control
```

```
C while TINS and TDEL are only used by TADD and not otherwise needed.
C The type of information stored in the buffer may vary from applicaton
C to application. This can be achieved by editing TADD. For example,
C if variable length integer data is to be stored, then BUFFER must be
C replaced by, INTGER, an integer array throughout. Likewise if BUFFER
C holds double precision or complex data, the statement that defines
C BUFFER in TADD must be changed accordingly. The program can also be
C used in other ways, for example, each node can refer to more than a
C single buffer. If this is done, both TSET and TADD must be modified
C in a rather obvious way to add the required multiple link-list and
C buffer arrays.
C
C ---------------------------------------------------------------------
C
C                              ***************
C                              ***  TSET   ***
C                              ***************
C
C The subroutine TSET must be called before inserting the first item
C into the tree. This call fixes the first 11 values of the array ID:
C
C       ID(-10): Number of nodes currently in the tree     ==> ID(-10)
C       ID(-9) : Maximum nodes in in tree                  ==> MXNODE
C       ID(-8) : Parent node of ID(-7), see next entry     ==> NT
C       ID(-7) : Node to be balanced                       ==> NA
C       ID(-6) : Parent node pointer                       ==> NQ
C       ID(-5) : Current node pointer                      ==> Determined
C       ID(-4) : Number of integers assigned the key       ==> NKEY
C       ID(-3) : Number of integers for key and data       ==> NSUM
C       ID(-2) : Position of the priority in a node        ==> NPR
C       ID(-1) : Position of the next available node       ==> Determined
C       ID( 0) : Root pointer (-1 for empty tree)          ==> Determined
C
C The subroutine TSET also initializes all entries of the array LLBUFF
C where the link-list information for BUFFER is stored. In particular,
C the first three which refer to the free space are assigned values:
C
C       LLBUFF(-2) : Tail of free space
C       LLBUFF(-1) : Head of free space
C       LLBUFF( 0) : Size of free space
C
C ---------------------------------------------------------------------
C
      SUBROUTINE TSET
C
      INTEGER ID(-10:*),LLBUFF(-2:*)
C
      ENTRY TSETMB(ID,MXNODE,NKEY,NDAT,LLBUFF,MXBUFF)
C
C Initialize link-list array values
C
      DO 10 J=1,MXBUFF-1
10       LLBUFF(J)=J+1
      LLBUFF(MXBUFF)=-1
C
```

```
      ENTRY TSETFB(ID,MXNODE,NKEY,NDAT,LLBUFF,MXBUFF)
C
C Initialize free space pointer/counter
C
      LLBUFF(-2)=MXBUFF
      LLBUFF(-1)=1
      LLBUFF(0)=MXBUFF
C
C Initialize WST parameters
C
      NSUM=NKEY+NDAT
      IMAX=MXNODE*(NSUM+3)
      ID(-10)=0
      ID(-9)=MXNODE
      ID(-8)=-1
      ID(-7)=-1
      ID(-6)=-1
      ID(-5)=0
      ID(-4)=NKEY
      ID(-3)=NSUM
      ID(-2)=NSUM+1
      ID(-1)=0
      ID(0)=-1
      DO 30 I=1,IMAX
30       ID(I)=0
      RETURN
      END
C ---------------------------------------------------------------------
C
C                              ************
C                              *** TADD ***
C                              ************
C
C The subroutine TADD allows the user to add elements to an existing
C database provided space is available. If it is not and the priority
C of the incoming element is greater than the lowest priority element
C currently in the database that element is eliminated to make room for
C the new one. The last scenario is repeated as necessary to accommodate
C the new element. The logic of the program is the following:
C
C Step 1) check if either the tree or buffer is full
C         if yes, check if the priority of incoming item is higher
C                 than that of the lowest priority node in the tree
C                 if yes, delete lowest priority node and goto Step 1)
C                 else RETURN
C         else    goto Step 2)
C Step 2) insert the incoming item into the database
C
C The entry TADDMB differs from entry TADDFB in that the size of the
C information stored in the buffer array may vary from node to node.
C To accommodate this feature, a link-list data structure is included.
C
C ---------------------------------------------------------------------
C
      SUBROUTINE TADD
```

```
C
      INTEGER NEWKEY(*),NEWDAT(*),ID(-10:*),LLBUFF(-2:*)
      REAL    BULOAD(*),BUFFER(*)                          ! *** storage type
C     REAL*8  BULOAD(*),BUFFER(*)                          ! *** storage type
      DATA    NF0/Z3FFFFFFF/
C
      ENTRY TADDFB(NEWKEY,NEWDAT,BULOAD,NOSIZE,NPBASE,ID,BUFFER,LLBUFF)
C
C ...define the number of keys in a node
C
      NKEY=ID(-4)
C
C ...calculate current priority
C
      NPCURR=NPBASE+ISHFT(NPBASE,8)
C
C ...check to see if the tree and BUFFERer can receive the new item
C
      IF ((ID(-10).LE.ID(-9)).AND.(LLBUFF(0).GE.NOSIZE)) THEN
         GOTO 300
C
C ...check if priority of new item is greater than lowest priority
C
      ELSE
100      IF (NPCURR.GT.IAND(ID(ID(-2)),NF0)) THEN
C
C Rearrange LLBUFF array which holds free space list information
C
            LLBUFF(-1)=ID(NKEY+1)
C
            CALL TDEL(ID)
C
            LLBUFF(0)=LLBUFF(0)+NOSIZE
            CALL TCHK(NEWKEY,ID)
            GOTO 300
C
C ...doing nothing when new priority is less than lowest priority
C
         ELSE
            RETURN
         ENDIF
      ENDIF
300   CONTINUE
C
C Load the key, data and its priority into the tree
C
C ...last key position of the current node
C
      IFIND=ID(-5)+NKEY
C
C ...load integer data into tree after the last key
C
      NEWDAT(1)=LLBUFF(-1)
      NEWDAT(2)=NOSIZE
      NDAT=ID(-3)-ID(-4)
```

```
      DO 400 IKK=1,NDAT
400   ID(IFIND+IKK)=NEWDAT(IKK)
C
C ...load real data into the BUFFERER and update LLBUFF
C
      DO 500 IKK=1,NOSIZE
         BUFFER(LLBUFF(-1))=BULOAD(IKK)
500      LLBUFF(-1)=LLBUFF(-1)+1
C
C ...reduce size of free space
C
      LLBUFF(0)=LLBUFF(0)-NOSIZE
C
C ...load the priority
C
      IFIND=ID(-5)+ID(-2)
      ID(IFIND)=NPCURR
C
C ...call TINS to complete (balance tree and restructure heap)
C
      CALL TINS(NEWKEY,ID)
      RETURN
C
      ENTRY TADDMB(NEWKEY,NEWDAT,BULOAD,NOSIZE,NPBASE,ID,BUFFER,LLBUFF)
C
C ...define the number of keys in a node
C
      NKEY=ID(-4)
C
C ...calculate current priority
C
      NPCURR=NPBASE+ISHFT(NPBASE,8)
C
C ...check to see if the tree and buffer can receive the new item
C
      IF ((ID(-10).LE.ID(-9)).AND.(LLBUFF(0).GE.NOSIZE)) THEN
         GOTO 301
C
C ...check if priority of new item is greater than lowest priority
C
      ELSE
101      IF (NPCURR.GT.IAND(ID(ID(-2)),NF0)) THEN
C
C Rearrange LLBUFF array which holds free space list information
C
            IX=ID(NKEY+1)
            IF (LLBUFF(0).EQ.0) THEN
               LLBUFF(0)=ID(NKEY+2)
               LLBUFF(-1)=IX
            ELSE
               LLBUFF(0)=LLBUFF(0)+ID(NKEY+2)
               LLBUFF(LLBUFF(-2))=IX
            ENDIF
            LLBUFF(-2)=IX
            DO 201 I=1,ID(NKEY+2)-1
```

```
                IX=LLBUFF(IX)
201             LLBUFF(-2)=IX
C
              CALL TDEL(ID)
C
              IF (LLBUFF(0).LT.NOSIZE) GOTO 101
              CALL TCHK(NEWKEY,ID)
              GOTO 301
C
C ...doing nothing when new priority is less than lowest priority
C
           ELSE
              RETURN
           ENDIF
        ENDIF
301     CONTINUE
C
C Load the key, data and its priority into the tree
C
C ...last key position of the current node
C
        IFIND=ID(-5)+NKEY
C
C ...load integer data into tree after the last key
C
        NEWDAT(1)=LLBUFF(-1)
        NEWDAT(2)=NOSIZE
        NDAT=ID(-3)-ID(-4)
        DO 401 IKK=1,NDAT
401     ID(IFIND+IKK)=NEWDAT(IKK)
C
C ...load real data into the BUFFER and update LLBUFF
C
        DO 501 IKK=1,NOSIZE
           BUFFER(LLBUFF(-1))=BULOAD(IKK)
501        LLBUFF(-1)=LLBUFF(LLBUFF(-1))
C
C ...reduce size of free space
C
        LLBUFF(0)=LLBUFF(0)-NOSIZE
C
C ...load the priority
C
        IFIND=ID(-5)+ID(-2)
        ID(IFIND)=NPCURR
C
C ...call TINS to complete (balance tree and restructure heap)
C
        CALL TINS(NEWKEY,ID)
        RETURN
        END
C ---------------------------------------------------------------------
C
C                                ************
C                                *** TCHK ***
```

```
C                              ************
C
C The subroutine TCHK constructs a weighted search tree or locates a
C specific node in a weighted search tree. The subroutine uses two
C arrays, NEWKEY and ID. A call to TCHK(,,) generates a search of ID for
C the key stored in NEWKEY. If the search is successful the priority of
C the node will be increased. If not successful, a normal return is
C generated and the position, ID(-5) where a new node can be inserted
C is given. After this subroutine is called, the position(s) where the
C key(s) of the new node are are to be assigned are ID(-5)+1, ID(-5)+2,
C ID(-5)+3, ... and the positions of the location and size of the new
C incoming data are ID(IPOS)+1 and ID(IPOS)+2, respectivaly, where
C IPOS = ID(-5)+ID(-4) and ID(-4) is the number of integer words set
C aside for the key.
C
C   RETURN 1 --> Successful search (key already in the tree,
C                ID(-5) is set to point to the located node).
C                The priority of the located node is updated.
C
C ---------------------------------------------------------------------
C
      SUBROUTINE TCHK(NEWKEY,ID,*)
C
      INTEGER NEWKEY(*),ID(-10:*)
      LOGICAL FLAG
C
C Integer data for a DEC system
C     DATA NF0,N2,N3/1073741823,-2147483648,-1073741824/
C Hexadecimal data for a IBM system
C
      DATA NF0,N2,N3/Z3FFFFFFF,Z80000000,ZC0000000/
      NKEY=ID(-4)
      NPR=ID(-2)
      NLC=NPR+1
      NRC=NPR+2
C
C Special case (empty tree)
C
      IF (ID(0).EQ.-1) THEN
         ID(-5)=0
         RETURN
      ENDIF
C
C Normal case (non-empty tree)
C
      NF=-1
      NA=ID(0)
      NP=ID(0)
      NQ=-1
100   IF (NP.NE.-1) THEN
C
C Check if the balance factor of NP is 0 or not
C
         IF (ID(NP+NPR).LT.0) THEN
            NA=NP
```

```
            NP=NQ
         ENDIF
         DO 101 I=1,NKEY
            IF (NEWKEY(I).LT.ID(NP+I)) THEN
               NQ=NP
               NO=ID(NP+NLC)
               IF (NO.EQ.-1) THEN
                  NP=-1
               ELSEIF (ID(NO+NRC).EQ.NP) THEN
                  IF (ISHFT(ID(NP+NPR),-30).EQ.2) THEN
                     NP=NO
                  ELSE
                     NP=-1
                  ENDIF
               ELSE
                  NP=NO
               ENDIF
               GOTO 100
            ELSEIF (NEWKEY(I).GT.ID(NP+I)) THEN
               NQ=NP
               NO=ID(NP+NLC)
               IF (NO.EQ.-1) THEN
                  NP=-1
               ELSEIF (ID(NO+NRC).EQ.NP) THEN
                  IF (ISHFT(ID(NP+NPR),-30).EQ.2) THEN
                     NP=-1
                  ELSE
                     NP=NO
                  ENDIF
               ELSE
                  NP=ID(NO+NRC)
               ENDIF
               GOTO 100
            ENDIF
101      CONTINUE
C
C If a match is found, increase the node priority, reconstruct the
C linear array, set the pointer to the node location, and RETURN 1
C
         NA=NP
         FLAG=.TRUE.
         ITEM=IAND(ID(NP+NPR),NF0)
         ITEMP=ITEM+ISHFT(ISHFT(ITEM,24),-16)
C
C Saturation condition for the priority
C
         IF (ITEMP.LE.NF0) THEN
            ID(NP+NPR)=ITEMP+IAND(ID(NP+NPR),N3)
            ITEMPR=ITEMP
         ELSE
            ITEMPR=ITEM
         ENDIF
         NL=ID(-1)
102      NB=NA+NA+NRC
         IF (NB.LT.NL) THEN
```

```
                NC=NB+NRC
                NBPR=IAND(ID(NB+NPR),NF0)
                NCPR=IAND(ID(NC+NPR),NF0)
                IF ((NBPR.LE.NCPR).OR.(NC.GE.NL)) THEN
                   IF (NBPR.LT.ITEMPR) THEN
                      IF (FLAG) THEN
                         NARC=ID(NA+NRC)
                         NALC=ID(NA+NLC)
                         IF (NARC.NE.-1) THEN
                            IF (ID(NARC+NLC).EQ.NA) THEN
                               ID(NARC+NLC)=NL
                            ELSE
                               IF (ID(ID(NARC+NRC)+NLC).EQ.NA) THEN
                                  ID(ID(NARC+NRC)+NLC)=NL
                               ELSE
                                  ID(ID(NARC+NLC)+NRC)=NL
                               ENDIF
                            ENDIF
                         ENDIF
                         IF (NALC.NE.-1) THEN
                            IF (ID(NALC+NRC).EQ.NA) THEN
                               ID(NALC+NRC)=NL
                            ELSE
                               ID(ID(NALC+NRC)+NRC)=NL
                            ENDIF
                         ENDIF
                         DO 105 I=1,NRC
105                         ID(NL+I)=ID(NA+I)
                         FLAG=.FALSE.
                      ENDIF
                      NBRC=ID(NB+NRC)
                      NBLC=ID(NB+NLC)
                      IF (NBRC.EQ.-1) THEN
                         ID(0)=NA
                      ELSE
                         IF (ID(NBRC+NLC).EQ.NB) THEN
                            ID(NBRC+NLC)=NA
                         ELSE
                            IF (ID(ID(NBRC+NRC)+NLC).EQ.NB) THEN
                               ID(ID(NBRC+NRC)+NLC)=NA
                            ELSE
                               ID(ID(NBRC+NLC)+NRC)=NA
                            ENDIF
                         ENDIF
                      ENDIF
                      IF (NBLC.NE.-1) THEN
                         IF (ID(NBLC+NRC).EQ.NB) THEN
                            ID(NBLC+NRC)=NA
                         ELSE
                            ID(ID(NBLC+NRC)+NRC)=NA
                         ENDIF
                      ENDIF
                      DO 109 I=1,NRC
                         ID(NA+I)=ID(NB+I)
109                   CONTINUE
```

```
                NA=NB
                GOTO 102
             ENDIF
          ELSE
             IF (NCPR.LT.ITEMPR) THEN
                IF (FLAG) THEN
                   NARC=ID(NA+NRC)
                   NALC=ID(NA+NLC)
                   IF (NARC.NE.-1) THEN
                      IF (ID(NARC+NLC).EQ.NA) THEN
                         ID(NARC+NLC)=NL
                      ELSE
                         IF (ID(ID(NARC+NRC)+NLC).EQ.NA) THEN
                            ID(ID(NARC+NRC)+NLC)=NL
                         ELSE
                            ID(ID(NARC+NLC)+NRC)=NL
                         ENDIF
                      ENDIF
                   ENDIF
                   IF (NALC.NE.-1) THEN
                      IF (ID(NALC+NRC).EQ.NA) THEN
                         ID(NALC+NRC)=NL
                      ELSE
                         ID(ID(NALC+NRC)+NRC)=NL
                      ENDIF
                   ENDIF
                   DO 112 I=1,NRC
112                   ID(NL+I)=ID(NA+I)
                   FLAG=.FALSE.
                ENDIF
                NCRC=ID(NC+NRC)
                NCLC=ID(NC+NLC)
                IF (NCRC.EQ.-1) THEN
                   ID(0)=NA
                ELSE
                   IF (ID(NCRC+NLC).EQ.NC) THEN
                      ID(NCRC+NLC)=NA
                   ELSE
                      IF (ID(ID(NCRC+NRC)+NLC).EQ.NC) THEN
                         ID(ID(NCRC+NRC)+NLC)=NA
                      ELSE
                         ID(ID(NCRC+NLC)+NRC)=NA
                      ENDIF
                   ENDIF
                ENDIF
                IF (NCLC.NE.-1) THEN
                   IF (ID(NCLC+NRC).EQ.NC) THEN
                      ID(NCLC+NRC)=NA
                   ELSE
                      ID(ID(NCLC+NRC)+NRC)=NA
                   ENDIF
                ENDIF
                DO 116 I=1,NRC
                   ID(NA+I)=ID(NC+I)
116             CONTINUE
```

```
                  NA=NC
                  GOTO 102
               ENDIF
            ENDIF
         ENDIF
118      IF (FLAG) GOTO 124
         NLRC=ID(NL+NRC)
         NLLC=ID(NL+NLC)
         IF (NLRC.EQ.-1) THEN
            ID(0)=NA
         ELSE
            IF (ID(NLRC+NLC).EQ.NL) THEN
               ID(NLRC+NLC)=NA
            ELSE
               IF (ID(ID(NLRC+NRC)+NLC).EQ.NL) THEN
                  ID(ID(NLRC+NRC)+NLC)=NA
               ELSE
                  ID(ID(NLRC+NLC)+NRC)=NA
               ENDIF
            ENDIF
         ENDIF
         IF (NLLC.NE.-1) THEN
            IF (ID(NLLC+NRC).EQ.NL) THEN
               ID(NLLC+NRC)=NA
            ELSE
               ID(ID(NLLC+NRC)+NRC)=NA
            ENDIF
         ENDIF
         DO 120 I=1,NRC
            ID(NA+I)=ID(NL+I)
            ID(NL+I)=0
120      CONTINUE
124      ID(-5)=NA
         RETURN 1
      ENDIF
      ID(-8)=NF
      ID(-7)=NA
      ID(-6)=NQ
      ID(-5)=ID(-1)+NRC
      RETURN
      END
C -------------------------------------------------------------------
C
C                              ************
C                              *** TINS ***
C                              ************
C
C The subroutine TINS is used to insert a new item into a WST. A call
C to TINS should follow a call to TCHK since TCHK tell where to insert
C the new item. After the item is placed in the WST, TINS rebalances the
C the tree and reconstruct the priority heap.
C
C -------------------------------------------------------------------
C
      SUBROUTINE TINS(NEWKEY,ID)
```

```
C
      INTEGER NEWKEY(*),ID(-10:*)
C
C Integer data for a DEC system
C     DATA NF0,N2,N3/1073741823,-2147483648,-1073741824/
C Hexadecimal data for a IBM  system
C
      DATA NF0,N2,N3/Z3FFFFFFF,Z80000000,ZC0000000/
      ID(-10)=ID(-10)+1
      NKEY=ID(-4)
      NPR=ID(-2)
      NLC=NPR+1
      NRC=NPR+2
C
C Special case (empty tree)
C
      IF (ID(0).EQ.-1) THEN
         ID(-1)=NRC
         ID(0)=0
C
C The first node position
C
         ID(NRC)=-1
         ID(NLC)=-1
         DO 10 I=1,NKEY
10          ID(I)=NEWKEY(I)
         RETURN
      ENDIF
C
C Normal case (non-empty tree)
C
      NY=ID(-5)
      NQ=ID(-6)
      NA=ID(-7)
      NF=ID(-8)
C
C Set the pointer to indicate the position of the new node and load key
C
      DO 130 I=1,NKEY
130      ID(NY+I)=NEWKEY(I)
      ID(NY+NLC)=-1
      IF (ID(NQ+NLC).EQ.-1) THEN
         ID(NY+NRC)=NQ
         ID(NQ+NLC)=NY
      ELSE
         DO 131 I=1,NKEY
            IF (NEWKEY(I).LT.ID(NQ+I)) THEN
               ID(NY+NRC)=ID(NQ+NLC)
               ID(NQ+NLC)=NY
               ID(NQ+NPR)=IAND(ID(NQ+NPR),NF0)
               GOTO 999
            ELSEIF (NEWKEY(I).GT.ID(NQ+I)) THEN
               ID(NY+NRC)=NQ
               ID(ID(NQ+NLC)+NRC)=NY
               ID(NQ+NPR)=IAND(ID(NQ+NPR),NF0)
```

```
                GOTO 999
             ENDIF
131       CONTINUE
          ENDIF
          DO 161 I=1,NKEY
             IF (NEWKEY(I).LT.ID(NA+I)) THEN
                NP=ID(NA+NLC)
                NB=NP
                IND=N2
                GOTO 200
             ELSEIF (NEWKEY(I).GT.ID(NA+I)) THEN
                NO=ID(NA+NLC)
                IF (ID(NO+NRC).EQ.NA) THEN
                   NP=NO
                ELSE
                   NP=ID(NO+NRC)
                ENDIF
                NB=NP
                IND=N3
                GOTO 200
             ENDIF
161       CONTINUE
200       IF (NP.NE.NY) THEN
             DO 211 I=1,NKEY
                IF (NEWKEY(I).LT.ID(NP+I)) THEN
                   ID(NP+NPR)=IAND(ID(NP+NPR),NF0)+N2
                   NP=ID(NP+NLC)
                   GOTO 200
                ELSEIF (NEWKEY(I).GT.ID(NP+I)) THEN
                   ID(NP+NPR)=IAND(ID(NP+NPR),NF0)+N3
                   NO=ID(NP+NLC)
                   IF (ID(NO+NRC).EQ.NP) THEN
                      NP=NO
                   ELSE
                      NP=ID(NO+NRC)
                   ENDIF
                   GOTO 200
                ENDIF
211          CONTINUE
          ENDIF
          IF (ID(NA+NPR).GE.0) THEN
             ID(NA+NPR)=IAND(ID(NA+NPR),NF0)+IND
             GOTO 999
          ELSEIF (IAND(ID(NA+NPR),N3).NE.IND) THEN
             ID(NA+NPR)=IAND(ID(NA+NPR),NF0)
             GOTO 999
          ENDIF
          IF (IND.EQ.N2) THEN
             IF (ISHFT(ID(NB+NPR),-30).EQ.2) THEN
                IF (ID(ID(NA+NLC)+NRC).EQ.NA) THEN
                   ID(NB+NRC)=ID(NA+NRC)
                   ID(ID(NB+NLC)+NRC)=NA
                   ID(NA+NLC)=-1
                   ID(NA+NRC)=NB
                ELSE
```

```
            NC=ID(ID(NB+NLC)+NRC)
            ID(NC+NRC)=ID(NB+NRC)
            ID(NB+NRC)=ID(NA+NRC)
            ID(ID(NB+NLC)+NRC)=NA
            ID(NA+NLC)=NC
            ID(NA+NRC)=NB
        ENDIF
        ID(NA+NPR)=IAND(ID(NA+NPR),NF0)
        ID(NB+NPR)=IAND(ID(NB+NPR),NF0)
    ELSE
        IF (ID(ID(NB+NLC)+NRC).EQ.NB) THEN
            NC=ID(NB+NLC)
        ELSE
            NC=ID(ID(NB+NLC)+NRC)
        ENDIF
        IF (ID(NC+NPR).GE.0) THEN
            ID(NC+NRC)=ID(NA+NRC)
            ID(NA+NLC)=-1
            ID(NA+NRC)=NC
            ID(NB+NLC)=-1
            ID(NC+NLC)=NB
            ID(NA+NPR)=IAND(ID(NA+NPR),NF0)
            ID(NB+NPR)=IAND(ID(NB+NPR),NF0)
        ELSE
            IF (ID(ID(NC+NLC)+NRC).EQ.NC) THEN
                IF (ISHFT(ID(NC+NPR),-30).EQ.2) THEN
                    ID(NA+NLC)=ID(NB+NRC)
                    ID(ID(NB+NLC)+NRC)=ID(NC+NLC)
                    ID(ID(NC+NLC)+NRC)=NB
                ELSE
                    ID(NA+NLC)=ID(NC+NLC)
                    ID(ID(NA+NLC)+NRC)=ID(NB+NRC)
                    ID(ID(NB+NLC)+NRC)=NB
                ENDIF
            ELSE
                ID(NA+NLC)=ID(ID(NC+NLC)+NRC)
                ID(ID(NA+NLC)+NRC)=ID(NB+NRC)
                ID(ID(NB+NLC)+NRC)=ID(NC+NLC)
                ID(ID(NC+NLC)+NRC)=NB
            ENDIF
            ID(NC+NLC)=NB
            ID(NC+NRC)=ID(NA+NRC)
            ID(NA+NRC)=NC
            ID(NB+NRC)=NA
            IF (ISHFT(ID(NC+NPR),-30).EQ.2) THEN
                ID(NA+NPR)=IAND(ID(NA+NPR),NF0)+N3
                ID(NB+NPR)=IAND(ID(NB+NPR),NF0)
            ELSE
                ID(NA+NPR)=IAND(ID(NA+NPR),NF0)
                ID(NB+NPR)=IAND(ID(NB+NPR),NF0)+N2
            ENDIF
            ID(NC+NPR)=IAND(ID(NC+NPR),NF0)
        ENDIF
        NB=NC
    ENDIF
```

```
ELSE
   IF (ISHFT(ID(NB+NPR),-30).EQ.3) THEN
      IF (ID(ID(NA+NLC)+NRC).EQ.NA) THEN
         ID(NA+NLC)=-1
         ID(NB+NRC)=ID(NA+NRC)
         ID(NA+NRC)=ID(NB+NLC)
         ID(NB+NLC)=NA
      ELSE
         NC=ID(NB+NLC)
         ID(NB+NLC)=NA
         ID(NB+NRC)=ID(NA+NRC)
         ID(ID(NA+NLC)+NRC)=NC
         ID(NA+NRC)=ID(NC+NRC)
         ID(NC+NRC)=NA
      ENDIF
      ID(NA+NPR)=IAND(ID(NA+NPR),NF0)
      ID(NB+NPR)=IAND(ID(NB+NPR),NF0)
   ELSE
      NC=ID(NB+NLC)
      IF (ID(NC+NPR).GE.0) THEN
         ID(NC+NRC)=ID(NA+NRC)
         ID(NC+NLC)=NA
         ID(NA+NLC)=-1
         ID(NA+NRC)=NB
         ID(NB+NLC)=-1
         ID(NB+NRC)=NC
         ID(NA+NPR)=IAND(ID(NA+NPR),NF0)
         ID(NB+NPR)=IAND(ID(NB+NPR),NF0)
      ELSE
         IF (ID(ID(NC+NLC)+NRC).EQ.NC) THEN
            IF (ISHFT(ID(NC+NPR),-30).EQ.3) THEN
               ID(ID(NA+NLC)+NRC)=NA
               ID(ID(NC+NLC)+NRC)=ID(NC+NRC)
               ID(NB+NLC)=ID(NC+NLC)
            ELSE
               ID(ID(NA+NLC)+NRC)=ID(NC+NLC)
               ID(ID(NC+NLC)+NRC)=NA
               ID(NB+NLC)=ID(NC+NRC)
            ENDIF
         ELSE
            ID(ID(NA+NLC)+NRC)=ID(NC+NLC)
            ID(NB+NLC)=ID(ID(NC+NLC)+NRC)
            ID(ID(NB+NLC)+NRC)=ID(NC+NRC)
            ID(ID(NC+NLC)+NRC)=NA
         ENDIF
         ID(NC+NLC)=NA
         ID(NC+NRC)=ID(NA+NRC)
         ID(NA+NRC)=NB
         ID(NB+NRC)=NC
         IF (ISHFT(ID(NC+NPR),-30).EQ.3) THEN
            ID(NA+NPR)=IAND(ID(NA+NPR),NF0)+N2
            ID(NB+NPR)=IAND(ID(NB+NPR),NF0)
         ELSE
            ID(NA+NPR)=IAND(ID(NA+NPR),NF0)
            ID(NB+NPR)=IAND(ID(NB+NPR),NF0)+N3
```

```
                ENDIF
                ID(NC+NPR)=IAND(ID(NC+NPR),NF0)
              ENDIF
              NB=NC
            ENDIF
          ENDIF
          IF (NF.EQ.-1) THEN
            ID(0)=NB
          ELSEIF (NA.EQ.ID(NF+NLC)) THEN
            ID(NF+NLC)=NB
          ELSE
            ID(ID(NF+NLC)+NRC)=NB
          ENDIF
999       CONTINUE
C
C Reconstruct the priority array
C
          NL=ID(-1)
          NRC2=NRC+NRC
1000      IF (MOD(NL,NRC2).EQ.0) THEN
            NA=NL/2-NRC
          ELSE
            NA=(NL-NRC)/2
          ENDIF
          IF (IAND(ID(NA+NPR),NF0).GT.IAND(ID(NY+NPR),NF0)) THEN
            NARC=ID(NA+NRC)
            NALC=ID(NA+NLC)
            IF (NARC.EQ.-1) THEN
              ID(0)=NL
            ELSE
              IF (ID(NARC+NLC).EQ.NA) THEN
                ID(NARC+NLC)=NL
              ELSE
                IF (ID(ID(NARC+NRC)+NLC).EQ.NA) THEN
                  ID(ID(NARC+NRC)+NLC)=NL
                ELSE
                  ID(ID(NARC+NLC)+NRC)=NL
                ENDIF
              ENDIF
            ENDIF
            IF (NALC.NE.-1) THEN
              IF (ID(NALC+NRC).EQ.NA) THEN
                ID(NALC+NRC)=NL
              ELSE
                ID(ID(NALC+NRC)+NRC)=NL
              ENDIF
            ENDIF
            DO 1400 I=1,NRC
              ID(NL+I)=ID(NA+I)
1400        CONTINUE
            NL=NA
            IF (NL.GT.0) GOTO 1000
          ENDIF
          NYRC=ID(NY+NRC)
          NYLC=ID(NY+NLC)
```

```
      IF (NYRC.EQ.-1) THEN
         ID(0)=NL
      ELSE
         IF (ID(NYRC+NLC).EQ.NY) THEN
            ID(NYRC+NLC)=NL
         ELSE
            IF (ID(ID(NYRC+NRC)+NLC).EQ.NY) THEN
               ID(ID(NYRC+NRC)+NLC)=NL
            ELSE
               ID(ID(NYRC+NLC)+NRC)=NL
            ENDIF
         ENDIF
      ENDIF
      IF (NYLC.NE.-1) THEN
         IF (ID(NYLC+NRC).EQ.NY) THEN
            ID(NYLC+NRC)=NL
         ELSE
            ID(ID(NYLC+NRC)+NRC)=NL
         ENDIF
      ENDIF
      DO 7000 I=1,NRC
         ID(NL+I)=ID(NY+I)
7000  CONTINUE
      ID(-1)=ID(-1)+NRC
      ID(-5)=NL
      RETURN
      END
C ---------------------------------------------------------------------
C
C                                 ************
C                                 *** TDEL ***
C                                 ************
C
C The subroutine TDEL deletes a node from a weighted search tree,
C keeping the binary tree balanced. The node deleted from the tree
C is the first element of the linear array ID. Since the linear array
C also has a heap structure, the root of a heap, which is the first
C element of an array, is the lowest priority element (minheap). The
C subroutine, TDEL, also manages the free space in the array BUFFER.
C The pointer array LLBUFF is used for this purpose.
C
C ---------------------------------------------------------------------
C
      SUBROUTINE TDEL(ID)
C
      INTEGER ID(-10:*)
      LOGICAL FLAG
C
C Integer data for a DEC system
C     DATA NF0,N2,N3/1073741823,-2147483648,-1073741824/
C Hexadecimal data for a IBM system
C
      DATA NF0,N2,N3/Z3FFFFFFF,Z80000000,ZC0000000/
C
C Initialize some integer constants
```

```
C
      NKEY=ID(-4)
      NPR=ID(-2)
      NLC=NPR+1
      NRC=NPR+2
C
C NA keeps track of most recent node with BF(0)
C NF is the parent of NA
C NQ follows NP through the tree
C
      NF=-1
      NA=ID(0)
      NP=ID(0)
      NQ=-1
      NR=-1
100   IF (NP.NE.-1) THEN
C
C Looking for the last node with BF(0) that is not a leaf
C
         IF ((ID(NP+NPR).GE.0).AND.(ID(NP+NLC).NE.-1)) THEN
            NA=NP
            NF=NQ
         ELSEIF (NQ.NE.-1) THEN
            IF (FLAG) THEN
               NO=ID(NQ+NLC)
               IF (ID(NO+NRC).EQ.NQ) THEN
                  NRCHILD=NO
               ELSE
                  NRCHILD=ID(NO+NRC)
               ENDIF
               IF ((ISHFT(ID(NQ+NPR),-30).EQ.3)
     *              .AND.(ID(NRCHILD+NPR).GE.0)) THEN
                  NA=NQ
                  NF=NR
               ENDIF
            ELSE
               IF ((ISHFT(ID(NQ+NPR),-30).EQ.2)
     *              .AND.(ID(ID(NQ+NLC)+NPR).GE.0)) THEN
                  NA=NQ
                  NF=NR
               ENDIF
            ENDIF
         ENDIF
         DO 101 I=1,NKEY
            IF (ID(I).LT.ID(NP+I)) THEN
               NR=NQ
               NQ=NP
               NP=ID(NP+NLC)
               FLAG=.TRUE.
               GOTO 100
            ELSEIF (ID(I).GT.ID(NP+I)) THEN
               NR=NQ
               NQ=NP
               NO=ID(NP+NLC)
               IF (ID(NO+NRC).EQ.NP) THEN
```

```
                  NP=NO
                ELSE
                  NP=ID(NO+NRC)
                ENDIF
                FLAG=.FALSE.
                GOTO 100
              ENDIF
101         CONTINUE
C
C A match is found
C
            GOTO 130
          ENDIF
C
C A match is not found
C
          WRITE(6,*) 'TREE IS EMPTY.'
          GOTO 9999
130       CONTINUE
          ID(-10)=ID(-10)-1
C
C Matched node does not have a child
C
          IF (ID(NP+NLC).EQ.-1) THEN
            IF (NQ.EQ.-1) THEN
              ID(0)=-1
              ID(-1)=0
              GOTO 9999
            ELSEIF (ID(NP+NRC).EQ.NQ) THEN
              IF (ID(NQ+NLC).EQ.NP) THEN
                ID(NQ+NLC)=-1
              ELSE
                ID(ID(NQ+NLC)+NRC)=NQ
              ENDIF
            ELSE
              ID(NQ+NLC)=ID(NP+NRC)
            ENDIF
C
C Matched node has only one child
C
          ELSEIF (ID(ID(NP+NLC)+NRC).EQ.NP) THEN
            NO=ID(NP+NLC)
            IF (ID(NP+NRC).EQ.-1) THEN
              ID(0)=ID(NO)
              ID(NO+NRC)=-1
              GOTO 999
            ELSE
              IF (ID(NQ+NLC).EQ.NP) THEN
                ID(NQ+NLC)=NO
              ELSE
                ID(ID(NQ+NLC)+NRC)=NO
              ENDIF
              ID(NO+NRC)=ID(NP+NRC)
            ENDIF
C
```

```
C Match node has both children
C
      ELSE
         NI=NQ
         NH=NP
         NG=ID(NP+NLC)
         IF (ID(NP+NPR).GE.0) THEN
            NA=NH
            NF=NI
         ELSEIF ((ISHFT(ID(NP+NPR),-30).EQ.3).AND.
     *           (ID(ID(NG+NRC)+NPR).GE.0)) THEN
            NA=NH
            NF=NI
         ENDIF
         IF (ID(NG+NLC).EQ.-1) THEN
            ID(ID(NG+NRC)+NRC)=NG
            ID(NG+NLC)=ID(NG+NRC)
            ID(NG+NRC)=ID(NP+NRC)
         ELSEIF (ID(ID(NG+NLC)+NRC).EQ.NG) THEN
           IF (ISHFT(ID(NG+NPR),-30).EQ.2) THEN
               ID(ID(NG+NLC)+NRC)=ID(NG+NRC)
               ID(ID(NG+NRC)+NRC)=NG
               ID(NG+NRC)=ID(NP+NRC)
            ELSE
               NH=NG
               NG=ID(NG+NLC)
               ID(NH+NLC)=-1
               ID(ID(NH+NRC)+NRC)=NG
               ID(NG+NLC)=NH
               ID(NG+NRC)=ID(NP+NRC)
            ENDIF
         ELSE
150         NI=NH
            NH=NG
            NG=ID(ID(NG+NLC)+NRC)
            IF(ID(NH+NPR).GE.0) THEN
               NA=NH
               NF=NI
            ELSEIF ((ISHFT(ID(NH+NPR),-30).EQ.2)
     *           .AND.(ID(ID(NH+NLC)+NPR).GE.0)) THEN
               NA=NH
               NF=NI
            ENDIF
            IF (ID(NG+NLC).EQ.-1) THEN
               ID(ID(NH+NLC)+NRC)=NH
            ELSEIF (ID(ID(NG+NLC)+NRC).EQ.NG) THEN
               IF (ISHFT(ID(NG+NPR),-30).EQ.2) THEN
                  ID(ID(NH+NLC)+NRC)=ID(NG+NLC)
                  ID(ID(NG+NLC)+NRC)=NH
               ELSE
                  NH=NG
                  NG=ID(NG+NLC)
                  ID(NH+NLC)=-1
               ENDIF
            ELSE
```

```
                GOTO 150
             ENDIF
             ID(ID(ID(NP+NLC)+NRC)+NRC)=NG
             ID(NG+NLC)=ID(NP+NLC)
             ID(NG+NRC)=ID(NP+NRC)
          ENDIF
          IF (NQ.EQ.-1) THEN
             ID(0)=NG
          ELSEIF (ID(NQ+NLC).EQ.NP) THEN
             ID(NQ+NLC)=NG
          ELSE
             ID(ID(NQ+NLC)+NRC)=NG
          ENDIF
          ID(NG+NPR)=IAND(ID(NG+NPR),NF0)+IAND(ID(NP+NPR),N3)
          IF (NA.EQ.NP) NA=NG
          IF (NF.EQ.NP) NF=NG
          DO 160 I=1,NKEY
160          ID(I)=ID(NG+I)
       ENDIF
C
C Balance the tree
C
500    CONTINUE
       IF (ID(NA+NLC).EQ.-1) THEN
          ID(NA+NPR)=IAND(ID(NA+NPR),NF0)
          GOTO 999
       ENDIF
       DO 180 I=1,NKEY
          IF (ID(I).LT.ID(NA+I)) THEN
             GOTO 190
          ELSEIF (ID(I).GT.ID(NA+I)) THEN
             NO=ID(NA+NLC)
             IF (NO.EQ.-1) THEN
                NP=-1
             ELSEIF (ID(NO+NRC).EQ.NA) THEN
                IF (ISHFT(ID(NA+NPR),-30).EQ.2) THEN
                   NP=-1
                ELSE
                   NP=NO
                ENDIF
             ELSE
                NP=ID(NO+NRC)
             ENDIF
             NB=NO
             IND=N2
             GOTO 300
          ENDIF
180    CONTINUE
190    NO=ID(NA+NLC)
       IF (NO.EQ.-1) THEN
          NP=-1
       ELSEIF (ID(NO+NRC).EQ.NA) THEN
          IF (ISHFT(ID(NA+NPR),-30).EQ.2) THEN
             NP=NO
          ELSE
```

```
            NP=-1
         ENDIF
      ELSE
         NP=NO
      ENDIF
      IF (ID(NO+NRC).EQ.NA) THEN
         NB=NO
      ELSE
         NB=ID(NO+NRC)
      ENDIF
      IND=N3
300   IF (ID(NA+NPR).GE.0) THEN
         ID(NA+NPR)=IAND(ID(NA+NPR),NF0)+IND
         GOTO 600
      ELSEIF (IAND(ID(NA+NPR),N3).NE.IND) THEN
         ID(NA+NPR)=IAND(ID(NA+NPR),NF0)
         GOTO 600
      ENDIF
C
C Rotate
C
      IF (IND.EQ.N2) THEN
         IF (ISHFT(ID(NB+NPR),-30).EQ.3) THEN
            IF (ID(ID(NB+NLC)+NRC).EQ.NB) THEN
               NC=ID(NB+NLC)
            ELSE
               NC=ID(ID(NB+NLC)+NRC)
            ENDIF
            IF (ID(NC+NPR).GE.0) THEN
               IF (ID(NB+NRC).EQ.NA) THEN
                  ID(NC+NRC)=ID(NA+NRC)
                  ID(NA+NLC)=-1
                  ID(NA+NRC)=NC
                  ID(NB+NLC)=-1
                  ID(NC+NLC)=NB
                  ID(NA+NPR)=IAND(ID(NA+NPR),NF0)
                  ID(NB+NPR)=IAND(ID(NB+NPR),NF0)
               ELSE
                  IDCL=ID(NC+NLC)
                  ID(NA+NLC)=ID(IDCL+NRC)
                  ID(ID(NA+NLC)+NRC)=ID(NB+NRC)
                  ID(ID(NB+NLC)+NRC)=IDCL
                  ID(IDCL+NRC)=NB
                  ID(NC+NLC)=NB
                  ID(NC+NRC)=ID(NA+NRC)
                  ID(NA+NRC)=NC
                  ID(NB+NRC)=NA
                  ID(NA+NPR)=IAND(ID(NA+NPR),NF0)
                  ID(NB+NPR)=IAND(ID(NB+NPR),NF0)
               ENDIF
            ELSE
               IDCL=ID(NC+NLC)
               IF (ID(IDCL+NRC).EQ.NC) THEN
                  IF (ISHFT(ID(NC+NPR),-30).EQ.2) THEN
                     ID(NA+NLC)=ID(NB+NRC)
```

```
                ID(ID(NB+NLC)+NRC)=IDCL
                ID(IDCL+NRC)=NB
            ELSE
                ID(NA+NLC)=IDCL
                ID(ID(NA+NLC)+NRC)=ID(NB+NRC)
                ID(ID(NB+NLC)+NRC)=NB
            ENDIF
        ELSE
            ID(NA+NLC)=ID(IDCL+NRC)
            ID(ID(NA+NLC)+NRC)=ID(NB+NRC)
            ID(ID(NB+NLC)+NRC)=IDCL
            ID(IDCL+NRC)=NB
        ENDIF
        ID(NC+NLC)=NB
        ID(NC+NRC)=ID(NA+NRC)
        ID(NA+NRC)=NC
        ID(NB+NRC)=NA
        IF (ISHFT(ID(NC+NPR),-30).EQ.2) THEN
            ID(NA+NPR)=IAND(ID(NA+NPR),NF0)+N3
            ID(NB+NPR)=IAND(ID(NB+NPR),NF0)
        ELSE
            ID(NA+NPR)=IAND(ID(NA+NPR),NF0)
            ID(NB+NPR)=IAND(ID(NB+NPR),NF0)+N2
        ENDIF
        ID(NC+NPR)=IAND(ID(NC+NPR),NF0)
    ENDIF
    NB=NC
ELSE
    IF (ID(NB+NRC).EQ.NA) THEN
        IF (ID(NB+NPR).GE.0) THEN
            NC=ID(ID(NB+NLC)+NRC)
            ID(NA+NLC)=NC
            ID(NC+NRC)=NA
        ELSE
            ID(NA+NLC)=-1
        ENDIF
        ID(NB+NRC)=ID(NA+NRC)
        ID(ID(NB+NLC)+NRC)=NA
        ID(NA+NRC)=NB
    ELSE
        NC=ID(ID(NB+NLC)+NRC)
        ID(NC+NRC)=ID(NB+NRC)
        ID(NB+NRC)=ID(NA+NRC)
        ID(ID(NB+NLC)+NRC)=NA
        ID(NA+NLC)=NC
        ID(NA+NRC)=NB
    ENDIF
    IF (ID(NB+NPR).GE.0) THEN
        ID(NA+NPR)=IAND(ID(NA+NPR),NF0)+N2
        ID(NB+NPR)=IAND(ID(NB+NPR),NF0)+N3
    ELSE
        ID(NA+NPR)=IAND(ID(NA+NPR),NF0)
        ID(NB+NPR)=IAND(ID(NB+NPR),NF0)
    ENDIF
ENDIF
```

```
ELSE
   IF (ISHFT(ID(NB+NPR),-30).EQ.2) THEN
      NC=ID(NB+NLC)
      IF (ID(NC+NPR).GE.0) THEN
         IF (ID(NA+NLC).EQ.NB) THEN
            ID(NC+NRC)=ID(NA+NRC)
            ID(NC+NLC)=NA
            ID(NA+NLC)=-1
            ID(NA+NRC)=NB
            ID(NB+NLC)=-1
            ID(NB+NRC)=NC
            ID(NA+NPR)=IAND(ID(NA+NPR),NF0)
            ID(NB+NPR)=IAND(ID(NB+NPR),NF0)
         ELSE
            IDCL=ID(NC+NLC)
            ID(ID(NA+NLC)+NRC)=IDCL
            ID(NB+NLC)=ID(IDCL+NRC)
            ID(ID(NB+NLC)+NRC)=ID(NC+NRC)
            ID(IDCL+NRC)=NA
            ID(NC+NLC)=NA
            ID(NC+NRC)=ID(NA+NRC)
            ID(NA+NRC)=NB
            ID(NB+NRC)=NC
            ID(NA+NPR)=IAND(ID(NA+NPR),NF0)
            ID(NB+NPR)=IAND(ID(NB+NPR),NF0)
         ENDIF
      ELSE
         IDCL=ID(NC+NLC)
         IF (ID(IDCL+NRC).EQ.NC) THEN
            IF (ISHFT(ID(NC+NPR),-30).EQ.3) THEN
               ID(ID(NA+NLC)+NRC)=NA
               ID(IDCL+NRC)=ID(NC+NRC)
               ID(NB+NLC)=IDCL
            ELSE
               ID(ID(NA+NLC)+NRC)=IDCL
               ID(IDCL+NRC)=NA
               ID(NB+NLC)=ID(NC+NRC)
            ENDIF
         ELSE
            ID(ID(NA+NLC)+NRC)=IDCL
            ID(NB+NLC)=ID(IDCL+NRC)
            ID(ID(NB+NLC)+NRC)=ID(NC+NRC)
            ID(IDCL+NRC)=NA
         ENDIF
         ID(NC+NLC)=NA
         ID(NC+NRC)=ID(NA+NRC)
         ID(NA+NRC)=NB
         ID(NB+NRC)=NC
         IF (ISHFT(ID(NC+NPR),-30).EQ.3) THEN
            ID(NA+NPR)=IAND(ID(NA+NPR),NF0)+N2
            ID(NB+NPR)=IAND(ID(NB+NPR),NF0)
         ELSE
            ID(NA+NPR)=IAND(ID(NA+NPR),NF0)
            ID(NB+NPR)=IAND(ID(NB+NPR),NF0)+N3
         ENDIF
```

```
                ID(NC+NPR)=IAND(ID(NC+NPR),NF0)
              ENDIF
              NB=NC
            ELSE
              IF (ID(NA+NLC).EQ.NB) THEN
                IF (ID(NB+NPR).GE.0) THEN
                  NC=ID(NB+NLC)
                  ID(NB+NRC)=ID(NA+NRC)
                  ID(NA+NLC)=NC
                  ID(NA+NRC)=ID(NC+NRC)
                  ID(NC+NRC)=NA
                ELSE
                  ID(NA+NLC)=-1
                  ID(NB+NRC)=ID(NA+NRC)
                  ID(NA+NRC)=ID(NB+NLC)
                ENDIF
                ID(NB+NLC)=NA
              ELSE
                NC=ID(NB+NLC)
                ID(NB+NLC)=NA
                ID(NB+NRC)=ID(NA+NRC)
                ID(ID(NA+NLC)+NRC)=NC
                ID(NA+NRC)=ID(NC+NRC)
                ID(NC+NRC)=NA
              ENDIF
              IF (ID(NB+NPR).GE.0) THEN
                ID(NA+NPR)=IAND(ID(NA+NPR),NF0)+N3
                ID(NB+NPR)=IAND(ID(NB+NPR),NF0)+N2
              ELSE
                ID(NA+NPR)=IAND(ID(NA+NPR),NF0)
                ID(NB+NPR)=IAND(ID(NB+NPR),NF0)
              ENDIF
            ENDIF
          ENDIF
          IF (NF.EQ.-1) THEN
            ID(0)=NB
          ELSEIF (NA.EQ.ID(NF+NLC)) THEN
            ID(NF+NLC)=NB
          ELSE
            ID(ID(NF+NLC)+NRC)=NB
          ENDIF
600       NF=NA
          NA=NP
          IF (NP.NE.-1) GOTO 500
999       CONTINUE
C
C Reconstruct the priority array
C
          NA=0
          NL=ID(-1)-NRC
          NLPR=IAND(ID(NL+NPR),NF0)
1000      NB=NA+NA+NRC
          IF (NB.LT.NL) THEN
            NC=NB+NRC
            NBPR=IAND(ID(NB+NPR),NF0)
```

```
            NCPR=IAND(ID(NC+NPR),NF0)
            IF ((NBPR.LE.NCPR).OR.(NC.GT.NL)) THEN
               IF (NBPR.LT.NLPR) THEN
                  NBRC=ID(NB+NRC)
                  NBLC=ID(NB+NLC)
                  IF (NBRC.EQ.-1) THEN
                     ID(0)=NA
                  ELSE
                     IF (ID(NBRC+NLC).EQ.NB) THEN
                        ID(NBRC+NLC)=NA
                     ELSE
                        IF (ID(ID(NBRC+NRC)+NLC).EQ.NB) THEN
                           ID(ID(NBRC+NRC)+NLC)=NA
                        ELSE
                           ID(ID(NBRC+NLC)+NRC)=NA
                        ENDIF
                     ENDIF
                  ENDIF
                  IF (NBLC.NE.-1) THEN
                     IF (ID(NBLC+NRC).EQ.NB) THEN
                        ID(NBLC+NRC)=NA
                     ELSE
                        ID(ID(NBLC+NRC)+NRC)=NA
                     ENDIF
                  ENDIF
                  DO 1400 I=1,NRC
                     ID(NA+I)=ID(NB+I)
1400              CONTINUE
                  NA=NB
                  GOTO 1000
               ENDIF
            ELSE
               IF (NCPR.LT.NLPR) THEN
                  NCRC=ID(NC+NRC)
                  NCLC=ID(NC+NLC)
                  IF (NCRC.EQ.-1) THEN
                     ID(0)=NA
                  ELSE
                     IF (ID(NCRC+NLC).EQ.NC) THEN
                        ID(NCRC+NLC)=NA
                     ELSE
                        IF (ID(ID(NCRC+NRC)+NLC).EQ.NC) THEN
                           ID(ID(NCRC+NRC)+NLC)=NA
                        ELSE
                           ID(ID(NCRC+NLC)+NRC)=NA
                        ENDIF
                     ENDIF
                  ENDIF
                  IF (NCLC.NE.-1) THEN
                     IF (ID(NCLC+NRC).EQ.NC) THEN
                        ID(NCLC+NRC)=NA
                     ELSE
                        ID(ID(NCLC+NRC)+NRC)=NA
                     ENDIF
                  ENDIF
```

```
                DO 2400 I=1,NRC
                   ID(NA+I)=ID(NC+I)
2400            CONTINUE
                NA=NC
                GOTO 1000
             ENDIF
          ENDIF
       ENDIF
       NLRC=ID(NL+NRC)
       NLLC=ID(NL+NLC)
       IF (NLRC.EQ.-1) THEN
          ID(0)=NA
       ELSE
          IF (ID(NLRC+NLC).EQ.NL) THEN
             ID(NLRC+NLC)=NA
          ELSE
             IF (ID(ID(NLRC+NRC)+NLC).EQ.NL) THEN
                ID(ID(NLRC+NRC)+NLC)=NA
             ELSE
                ID(ID(NLRC+NLC)+NRC)=NA
             ENDIF
          ENDIF
       ENDIF
       IF (NLLC.NE.-1) THEN
          IF (ID(NLLC+NRC).EQ.NL) THEN
             ID(NLLC+NRC)=NA
          ELSE
             ID(ID(NLLC+NRC)+NRC)=NA
          ENDIF
       ENDIF
       DO 7000 I=1,NRC
          ID(NA+I)=ID(NL+I)
          ID(NL+I)=0
7000   CONTINUE
       ID(-1)=NL
9999   RETURN
       END
C ---------------------------------------------------------------------
C
C                              ***************
C                              ***  TOUT   ***
C                              ***************
C
C The subroutine TOUT traverses the weighted search tree in order to
C specific nodes and sends the outputs to an output devise designated
C by NFILE. The traversal is done in ascending order if NAD=1 and in
C descending order otherwise. A description of the arguments follows:
C
C   NFILE = File number assigned to output device where the results are
C           to be written. In the special case NFILE = 0, TOUT does not
C           output anything, however after upon returning ID(-5) points
C           to the current node position.
C     NAD = 1 if the traversal is to be in ascending order, otherwise
C           it is done in descending order.
C     MIN = Number of the first node that is to be retrieved.
```

```
C     MAX = Number of the last node that is to be retrieved.
C   NSTEP = Step size; nodes that are integer multiples of NSTEP beyond
C           MIN up to MAX will be retrieved.
C      ID = Name of the tree array that is to be traversed.
C
C An internal array called STACK is used to store information about the
C path traversed. It is dimensioned 32 since this is the maximum height
C the tree can have.  This limit is set by the fact that 2**32-1 is the
C largest integer pointer that can be used on a 32 bit machine.
C
C ---------------------------------------------------------------------
C
      SUBROUTINE TOUT(NFILE,NAD,MIN,MAX,NSTEP,ID)
C
      INTEGER ID(-10:*),STACK(32)
      NSUM=ID(-3)
      NLC=NSUM+2
      NRC=NLC+1
      I=0
      M=0
      NUM=MIN
      NODE=ID(0)
      IF (NAD.EQ.1) THEN
200      IF (NODE.NE.-1) THEN
            I=I+1
            STACK(I)=NODE
            NO=ID(NODE+NLC)
            IF (NO.EQ.-1) THEN
               NODE=-1
            ELSEIF (ID(NO+NRC).EQ.NODE) THEN
               IF (ISHFT(ID(NODE+NPR),-30).EQ.2) THEN
                  NODE=NO
               ELSE
                  NODE=-1
               ENDIF
            ELSE
               NODE=NO
            ENDIF
            GOTO 200
300         M=M+1
            ID(-5)=NODE
            IF ((M.EQ.NUM).OR.(M.EQ.MAX)) THEN
               IF (NFILE.GT.0) WRITE(NFILE,1000)
     *            M,(ID(NODE+II),II=1,NSUM)
               IF (M.EQ.MAX) GOTO 999
               NUM=NUM+NSTEP
            ENDIF
            NO=ID(NODE+NLC)
            IF (NO.EQ.-1) THEN
               NODE=-1
            ELSEIF (ID(NO+NRC).EQ.NODE) THEN
               IF (ISHFT(ID(NODE+NPR),-30).EQ.2) THEN
                  NODE=-1
               ELSE
                  NODE=NO
```

```
                ENDIF
              ELSE
                NODE=ID(NO+NRC)
              ENDIF
              GOTO 200
            ENDIF
            IF (I.NE.0) THEN
              NODE=STACK(I)
              I=I-1
              GOTO 300
            ENDIF
          ELSE
400         IF (NODE.NE.-1) THEN
              I=I+1
              STACK(I)=NODE
              NO=ID(NODE+NLC)
              IF (NO.EQ.-1) THEN
                NODE=-1
              ELSEIF (ID(NO+NRC).EQ.NODE) THEN
                IF (ISHFT(ID(NODE+NPR),-30).EQ.2) THEN
                  NODE=-1
                ELSE
                  NODE=NO
                ENDIF
              ELSE
                NODE=ID(NO+NRC)
              ENDIF
              GOTO 400
500           M=M+1
              ID(-5)=NODE
              IF ((M.EQ.NUM).OR.(M.EQ.MAX)) THEN
                IF (NFILE.GT.0) WRITE(NFILE,1000)
     *            M,(ID(NODE+II),II=1,NSUM)
                IF (M.EQ.MAX) GOTO 999
                NUM=NUM+NSTEP
              ENDIF
              NO=ID(NODE+NLC)
              IF (NO.EQ.-1) THEN
                NODE=-1
              ELSEIF (ID(NO+NRC).EQ.NODE) THEN
                IF (ISHFT(ID(NODE+NPR),-30).EQ.2) THEN
                  NODE=NO
                ELSE
                  NODE=-1
                ENDIF
              ELSE
                NODE=NO
              ENDIF
              GOTO 400
            ENDIF
            IF (I.NE.0) THEN
              NODE=STACK(I)
              I=I-1
              GOTO 500
            ENDIF
```

```
      ENDIF
      WRITE(NFILE,*)
      WRITE(NFILE,*)'  WARNING:  TOUT SEARCH HAS GONE OUT OF BOUNDS!'
999   RETURN
1000  FORMAT(3X,I6,':',2X,14I5/(12X,14I5))
      END
```

APPENDIX B

CODES OF SPARSE MATRIX COMPUTATIONS

B.1 Algorithms for Sparse Matrix Computations

B.1a Matrix Transpose

The procedure TRANSPOSE takes the transpose of matrix A and stores the result in matrix B.

```
procedure TRANSPOSE(EA,SA,EB,SB)
// bucket sort algorithm simulation with 1-tuple using a two-dimensional array //
        for i ← 1 to SA(0,2) do
           PNT(i) ← 1
        end
        i ← 1; count ← 0
        while i ≤ SA(0,3) do
           for j ← SA(i,2) to SA(i,3)
              count ← count + 1
              QUEUE(j,PNT(j),1) ← EA(count); QUEUE(j,PNT(j),2) ← SA(i,1)
              PNT(j) ← PNT(j) + 1
           end
           i ← i + 1
        end (while)
// end bucket sort construction//
        j ← k ← p ← 1; seg ← 0; last ← -1;
        while p ≤ count do
           if QUEUE(j,k,2) ≠ 0 then [
              EB(l) ← QUEUE(j,k,1)
              if last + 1 = QUEUE(j,k,2) then [
                 last ← last + 1; SB(seg,3) ← last]
              else [
                 last ← QUEUE(j,k,2); seg ← seg + 1;
                 SB(seg,1) ← j; SB(seg,2) ← last; SB(seg,3) ← last]
              k ← k + 1; p ← p + 1]
           else [
              k ← 1; j ← j + 1; last ← -1]
        end (while)
        SB(0,1) ← SA(0,2); SB(0,2) ← SA(0,1); SB(0,3) ← seg
end TRANSPOSE
```

B.1b Matrix Multiplication

Before giving the a matrix multiplication procedure, it is useful to present a sub-procedure called *STORESUM*. The purpose of this sub-procedure is to store the results C_{ij} in the proper position in the sequence of non-zero elements in EC and to update SC.

```
procedure STORESUM(EC,SC,last,flag,count,row,col,sum)
        if flag then last ← -1
        if sum ≠ 0 then [
           count ← count + 1
           EC(count) ← sum
           if last + 1 = col then [
              last ← col; SC(noseg,3) ← col]
           else [
              noseg ← noseg + 1
              last ← col
              SC(noseg,1) ← row; SC(noseg,2) ← col; SC(noseg,3) ← col]
           sum ← 0]
        flag ← false
end STORESUM
```

The procedure *MMULT* multiplies matrix B by A to obtain the product matrix C.

```
procedure MMULT(EA,SA,EB,SB,EC,SC)
        call TRANSPOSE(EB,SB,EB^T,SB^T)
        i ← p ← p_origin ← row_origin ← 1; sum ← 0; count ← 0; noseg ← 0;
        row ← SA(i,1); i2 ← SA(i,2);
        while i ≤ SA(0,3) do
           j ← k ← 1; flag ← true; col ← SB^T(1,1); j2 ← SB^T(1,2)
           while j ≤ SB^T(0,3) + 1 do
              case
                 : SA(i,1) ≠ row :
                    call STORESUM(EC,SC,last,flag,count,row,col,sum)
                    i=row_origin; p ← p_origin; i2 ← SA(i,2)
                    if SB^T = col and j2 > SB^T(j,2) then
                       [k ← k + SB^T(j,3) - j2 + 1; j ← j + 1]
```

```
    while SB^T(j,1) = col do
       k ← k + SB^T(j,3) - SB^T(j,2) + 1; j ← j + 1
    end (while)
    j2 ← SB^T(j,2); col ← SB^T(j,1)
 : SB^T(j,1) ≠ col:
    call STORESUM(EC,SC,last,flag,count,row,col,sum)
    i ← row_origin; i2 ← SA(i,2); p ← p_origin; col ← SB^T(j,1)
 : i2 > j2 :
    if i2 ≤ SB^T(j,3) then [
       k ← k + i2 - j2
       if SA(i,3) > SB^T(j,3) then [
          for m ← i2 to SB^T(j,3) do
             sum ← sum + EA(l) × EB^T(k); p ← p + 1; k ← k + 1
          end
          j ← j + 1; i2 ← m; j2 ← SB^T(j,2)]
       elseif SA(i,3) = SB^T(j,3) then [
          for m ← i2 to SA(i,3) do
             sum ← sum + EA(l) × EB^T(k); p ← p + 1; k ← k + 1
          end
          i ← i + 1; j ← j + 1; i2 ← SA(i,2); j2 ← SB^T(j,2)]
       else [
          for m ← i2 to SA(i,3) do
             sum ← sum + EA(l) × EB^T(k); p ← p + 1; k ← k + 1
          end
          j2 ← m; i ← i + 1; i2 ← SA(i,2)]]
    else [
       if j2 > SB^T(j,2) then k ← k + SB^T(j,3) - j2 + 1
       else k ← k + SB^T(j,3) - SB^T(j,2) + 1
       j ← j + 1; j2 ← SB^T(j,2)]
 : SA(i,3) ≥ j2 :
    l ← p + j2 - i2;
    if SA(i,3) > SB^T(j,3) then [
       for m ← j2 to SB^T(j,3) do
          sum ← sum + EA(l) × EB^T(k); p ← p + 1; k ← k + 1
       end
       i2 ← m; j ← j + 1; j2 ← SB^T(j,2)]
    elseif SA(i,3)= SB^T(j,3) then [
       for m ← j2 to SB^T(j,3) do
          sum ← sum + EA(l) × EB^T(k); p ← p + 1; k ← k + 1
       end
       i ← i + 1; j ← j + 1; i2 ← SA(i,2); j2 ← SB^T(j;2)]
    else [
       for m ← j2 to SA(i,3) do
          sum ← sum + EA(l) × EB^T(k); p ← p + 1; k ← k + 1
       end
       i ← i + 1; j2 ← m; i2 ← SA(i,2)] ]
 : else :
```

```
                    if i2 > SA(i,2)) then p ← p + SA(i,3) - i2 + 1
                    else p ← p + SA(i,3) - SA(i,2) + 1
                    i ← i + 1; i2 ← SA(i,2)
                end (case)
            end
            if SA(i,1) = row and i2 > SA(i,2) then [
                l ← p + SA(i,3) - i2 + 1; i ← i + 1]
            while SA(i,1) = row do
                l ← p + SA(i,3) - SA(i,2) + 1; i ← i + 1
            end
            row_origin ← i; p_origin ← p; row ← SA(i,1); i2 ← SA(i,2)
        end
        SC(0,1) ← SA(0,1); SC(0,2) ← SB(0,2); SC(0,3) ← noseg
end MMULT
```

B.1c Miscellaneous Subprocedures

Two subroutines, one called *PACKING* for converting a conventional two-dimensional array into the new representation, and another called *UNPACKING* for the inverse process, are included for convenience.

```
procedure PACKING(A,ni,nj,EA,SA)
        count ← 0; noseg ← 0
        for i ← 1 to ni do
            last ← -1
            for j ← 1 to nj do
                if A(i,j) ≠ 0 then [
                    count ← count + 1
                    EA(count) ← A(i,j)
                    if last + 1 = j then [
                        last ← j; SA(noseg,3) ← j]
                    else [
                        last ← j; noseg ← noseg + 1
                        SA(noseg,1) ← i; SA(noseg,2) ← j; SA(noseg,3) ← j]]
            end
        end
        SA(0,1) ← ni; SA(0,2) ← nj; SA(0,3) ← noseg
end PACKING

procedure UNPACKING(EA,SA,A)
```

```
            count ← 1
            for i ← 1 to SA(0,3) do
               for j ← SA(i,2) to SA(i,3) do
                  A(SA(i,1),j) ← EA(count)
                  count ← count + 1
               end
            end
end UNPACKING
```

B.2 Program Summary

Title of program: SMM

Computer: IBM 3090/600E

Operating system: MVS/XA (Version 3, Release 13)

Programming language used: FORTRAN

High speed storage required: The five routines that comprise the sparse matrix multiplication package (SMM) require a total of 8070 bytes of memory space on the IBM 3090 mainframe computer. The main program and array storage requirements are over and above this base amount.

Peripherals used: none

No. of lines in program: 575 (156 in sample DRIVER, 419 in SMM package)

Keywords: sparse matrix, data structures, matrix multiplication, matrix transpose.

Nature of the physical problem

Sparse matrix multiplication [1-3] often arises in scientific computations. Since a sparse matrix includes many zero elements, the multiplication should not be handled in the same was as for dense matrices. The standard matrix multiplication algorithm for $n \times n$ factor matrices, represented in the usual two-dimensional array form, takes $O(n^3)$ time [3]. This means that when the factor matrices are very large, e.g., 1000's $\times$ 1000's, not only will the computation time be excessively long but the demands on storage can strain even the biggest of modern computers. Developing efficient data structures and algorithms for the multiplication of sparse matrices is therefore very important.

The new data structure and algorithm for sparse matrices that is presented in this paper is more time and space efficient than the existing methods if the sparse matrices contain nonzero elements which are partially or fully adjacent to one another as in band or triangular matrices. Space complexity is better than that of the existing algorithms when the number of the groups of adjacent non-zero elements is less than two-thirds of the total number of nonzero elements. Time complexity is better or much better than that of existing algorithms depending on the number of groups of nonzero adjacent elements in the factor matrices.

Method of solution

The sparse matrix multiplication problem is addressed by introducing a space efficient data structure for representing the matrices and a multiplication algorithm that can be easily vectorized that is based on the new representation. The new structure represents a sparse matrix with two arrays, one that contains only the nonzero elements and the another integer array that holds information on the storage of matrix segments, which are the sets of adjacent nonzero elements in a row. For the multiplication of two matrices, $A \times B$, the transpose B^T is determined first and then the segments of A are compared to those of B^T to calculate segment overlaps. Details on how this is done are presented in the text.

Restrictions on the complexity of the problem

There are no restrictions on the factor matrices in the product. However, depending on the size of the matrices, memory overflow could be a problem.

Typical running time: For 300×300 band matrices with a bandwidth of 31, the matrix multiplication operation takes 0.74 sec on an IBM 3090/600E

Unusual features of the program

The routines in the package are generic and require no modification except for possible changes in the implicit statements that specify the character of the factor matrices.

References

[1] J. K. Cullum and R. A. Willoughby, "*Lanczos Algorithms for Large Symmetric Eigenvalue Computations*," Vol. 1, Birkhauser Boston, Germany, 1985

[2] G. H. Golub and C. F. Van Loan, "*Matrix Computations*," Second Ed., The Johns Hopkins University Press, Baltimore, MD, 1989.

[3] E. Horowitz and S.SAhni, "*Fundamentals of Data Structures*," Computer Science Press, Rockville, MD, 1983.

[4] M. J. Quinn, "*Designing Efficient Algorithms for Parallel Computers*," McGraw-Hill, NY, 1987.

B.3 Sample Program Using the Sparse Matrix Multiplication Package

```
C ---------------------------------------------------------------------
C
C             ************************************************************
C             ****           SAMPLE APPLICATION PROGRAM                ****
C             ****                    using the                        ****
C             ****  SPARSE MATRIX MULTIPLICATION (SMM) PACKAGE         ****
C             ************************************************************
C
C ---------------------------------------------------------------------
C
C Authors: Soon Park, J. P. Draayer and S.-Q.Zheng
C          Departments of Computer Science/Physics and Astronomy
C          Louisiana State University
C          Baton Rouge LA
C          USA 70803-4001
C
C                  BITNET:  PHDRYR @ LSUMVS or LSUVM
C                  TELEX:   559184
C                  PHONE:   USA-504-388-2261
C                  FAX:     USA-504-388-5855
C
C ---------------------------------------------------------------------
C
C Updates: 03/91 Original from a FORTRAN code written by Soon Park
C
C ---------------------------------------------------------------------
C
C A sample application program which consists of a driver only:
C
C   1.  DRIVER -->  A main program illustrating the use of routines
C                   from the sparse matrix multiplication package.
C
C The accompanying SPARSE MATRIX MULTIPLICATION package (SMM) that
C includes a total of five subroutines:
C
C   1.  MMULT      -->  Multiply two matrices when both are represented
C                       in the new sparse matrix form.
C   2.  TRANSPOSE  -->  Take the transpose of a matrix that is given in
C                       the new representation.
C   3.  PACKING    -->  Pack a matrix given in standard two-dimensional
C                       representation into the new sparse matrix form.
C   4.  UNPACKING  -->  Convert a matrix given in the new form into a
C                       standard two-dimensional representation.
C   5.  PRINTOUT   -->  Print out the elements of a matrix that is given
C                       in the standard two-dimensional form.
C
C ---------------------------------------------------------------------
C
C                           ***************
C                           *** DRIVER ***
C                           ***************
```

```
C
C This is an elementary program designed to test the routines in the
C sparse matrix multiplication package. DRIVER multiplies two sparse
C matrices which are the 300 x 300 band matrices with a bandwith of
C 31. The process used by DRIVER is the following:
C
C Step 1) Generate two 300 x 300 band matrices, each with a bandwith
C         of 31, in the standard two-dimensional array form.
C Step 2) Pack the two dimensional arrays into the new sparse matrix
C         representation using the PACKING algorithm.
C Step 3) Multiply the two matrices using the new MMULT algorithm for
C         multiplying matrices given in the new form.
C Step 4) Print out ranges of elements in a matrix after converting
C         it via UNPACKING it to standard two-dimensional form.
C
C ------------------------------------------------------------------
C
C Main Program
C
      PARAMETER (IDIM=300,JDIM=300,NELE=18300,NSEG=300,NSPO=NSEG+1)
      DIMENSION A(IDIM,JDIM),B(IDIM,JDIM),C(IDIM,JDIM)
      DIMENSION EA(NELE),EB(NELE),EC(NELE),ET(NELE)
      DIMENSION NSA(0:NSPO,3),NSB(0:NSPO,3),NSC(0:NSPO,3),NST(0:NSPO,3)
      DIMENSION QUE(JDIM,IDIM),KUE(JDIM,IDIM),NPNT(JDIM)
C
C ... generate two sparse matrices (band matrices)
C
      DO I=1,IDIM
         DO J=1,JDIM
           IF (ABS(I-J).LE.15) A(I,J)=1
         ENDDO
      ENDDO
      DO I=1,IDIM
         DO J=1,JDIM
           IF (ABS(I-J).LE.15) B(I,J)=1
         ENDDO
      ENDDO
C
C ... pack matrices into new representation form
C
      CALL PACKING(A,EA,NSA,IDIM,JDIM,NSPO)
      CALL PACKING(B,EB,NSB,IDIM,JDIM,NSPO)
C
C ... check cpu time for VAX
C     T1=SECNDS(0.0)
C
C ... check cpu time for IBM 3090/600E
C     CALL STKLOK
C
C ... multiply matrices using the new representation
C
      CALL MMULT(EA,NSA,EB,NSB,EC,NSC,ET,NST,
     *           QUE,KUE,NPNT,IDIM,JDIM,NSPO)
C
C ... check cpu time for VAX
```

```
C      TIME=SECNDS(T1)
C
C ... check cpu time for IBM 3090/600E
C      CALL KLOK(III)
C      TIME=.01*III
C
C      WRITE(6,*) 'TIME ===> ',TIME
C
C ... unpack resultant into two-dimensional array form
C
       CALL UNPACKING(EC,NSC,C,IDIM,JDIM,NSPO)
C
C ... report the results of the procedure to the user
C
       WRITE(6,*)
       WRITE(6,*) ' *** PROGRAM RAN SUCCESSIVELY! ***'
       WRITE(6,*)
       WRITE(6,*) '      *************************'
       WRITE(6,*) '      **   OUTPUT OPTIONS    **'
       WRITE(6,*) '      *************************'
       WRITE(6,*)
600    CONTINUE
       WRITE(6,*)
       WRITE(6,*) ' VIEW THE MATRIX?  ... SELECT AN OPTION:'
       WRITE(6,*)
       WRITE(6,*) '     1 = YES'
       WRITE(6,*) '     2 = QUIT OR STOP LOOKING'
       WRITE(6,*) '     ?'
C
       READ(5,*,END=999) NAD
       WRITE(6,1200) NAD
       IF (NAD.GE.2) GO TO 999
       WRITE(6,*) 'ENTER THE START POSITION OF ROW'
       READ(5,*,END=999) IDIM1
       WRITE(6,1200) IDIM1
       WRITE(6,*) 'ENTER THE FINAL POSITION OF ROW'
       READ(5,*,END=999) IDIM2
       WRITE(6,1200) IDIM2
       WRITE(6,*) 'ENTER THE START POSITION OF COLUMN'
       READ(5,*,END=999) JDIM1
       WRITE(6,1200) JDIM1
       WRITE(6,*) 'ENTER THE FINAL POSITION OF COLUMN'
       READ(5,*,END=999) JDIM2
       WRITE(6,1200) JDIM2
       CALL PRINTOUT(C,IDIM,JDIM,IDIM1,IDIM2,JDIM1,JDIM2)
       GOTO 600
 999   CONTINUE
       STOP
1200   FORMAT(' ====> SELECTED VALUE:',I5,2X,'<====='/)
       END
C ---------------------------------------------------------------------
C
C                          ************************
C                          ***    SMM PACKAGE   ***
C                          ************************
```

```
C
C                 ------------------------------------
C
C          ******************************************************
C          ***     SPARSE MATRIX MULTIPLICATION PACKAGE       ***
C          ******************************************************
C
C ---------------------------------------------------------------------
C
C Authors: Soon Park, J. P. Draayer and S.-Q. Zheng
C          Departments of Computer Science/Physics and Astronomy
C          Louisiana State University
C          Baton Rouge LA
C          USA 70803-4001
C
C                  BITNET:  PHDRYR @ LSUMVS or LSUVM
C                  TELEX:   559184
C                  PHONE:   USA-504-388-2261
C                  FAX:     USA-504-388-5855
C
C ---------------------------------------------------------------------
C
C Updates: 03/91 Original from a FORTRAN code written by Soon Park
C
C ---------------------------------------------------------------------
C
C General comments on the package:
C
C   This matrix package is written in FORTRAN because most scientific
C   applications require FORTRAN compatibility and the largest number
C   of existing scientific subroutine libraries are available in this
C   form. The SMM package multiplies sparse matrices A to B to get C.
C
C ---------------------------------------------------------------------
C
C This package consists of five subroutines:
C
C   1.  MMULT       -->  Multiply two matrices when both are represented
C                        in the new sparse matrix form.
C   2.  TRANSPOSE   -->  Take the transpose of a matrix that is given in
C                        the new representation.
C   3.  PACKING     -->  Pack a matrix given in standard two-dimensional
C                        representation into the new sparse matrix form.
C   4.  UNPACKING   -->  Convert a matrix given in the new form into a
C                        standard two-dimensional representation.
C   5.  PRINTOUT    -->  Print out the elements of a matrix that is given
C                        in the standard two-dimensional form.
C
C ---------------------------------------------------------------------
C
C                            *************
C                            *** MMULT ***
C                            *************
C
C The subroutine MMULT multiplies two sparse matrices (EA,NSA) and
```

```
C (EB,NSB) which are given in the new sparse matrix representation.
C The results is (EC,NSC). The new sparse matrix representation of
C matrix X consists of two arrays EX(1:t) and NSX(0:s+1,1:3), where
C t is the number of nonzero elements in X and s is the number of
C segments in X in row-major order. For bookkeeping purposes, the
C size of NSX must be one bigger than the number of segments. The
C NSX array contains s+1 entries with each entry a 3-tuple of type
C (NSX(r,1),NSX(r,2),NSX(r,3)). NSX(0,1) and NSX(0,2) are the number
C of rows and columns in the matrix X, respectively, while NSX(0,1)
C specifies the number of segments in X. Specifically, for an m x n
C matrix X with s segments NSX(0,1) = m, NSX(0,2) = n, NSX(0,3) = s.
C The r-th entry, r > 0, specifies information on the segments. For
C example if segment(j,k) is in row i then NSX(r,1) = i, NSX(r,2) =
C j, NSX(r,3) = k. Since nonzero elements and segments of the matrix
C X are arranged in EX in a unique linear order, the indices of the
C nonzero elements can be calculated by a linear scan of NSX. Here
C the pair (EX,NSX) is used to denote X in the new representation.
C
C ---------------------------------------------------------------------
      SUBROUTINE MMULT(EA,NSA,EB,NSB,EC,NSC,ET,NST,
     *                 QUE,KUE,NPNT,IDIM,JDIM,NSPO)
      DIMENSION EA(*),EB(*),EC(*),ET(*)
      DIMENSION NSA(0:NSPO,3),NSB(0:NSPO,3),NSC(0:NSPO,3),NST(0:NSPO,3)
      DIMENSION QUE(JDIM,IDIM),KUE(JDIM,IDIM),NPNT(JDIM)
      CALL TRANSPOSE(EB,NSB,ET,NST,QUE,KUE,NPNT,IDIM,JDIM,NSPO)
      I=1
      L=1
      L0=1
      LOW0=1
      LOW=NSA(1,1)
      I2=NSA(1,2)
      SUM=0.0
      KOUNT=0
      NOSEG=0
      NT=NST(0,3)+1
      DO WHILE (I.LE.NSA(0,3))
         J=1
         K=1
         KOL=NST(1,1)
         J2=NST(1,2)
         LAST=-1
         DO WHILE (J.LE.NT)
            IF (NSA(I,1).NE.LOW) THEN
               IF (SUM.NE.0.0) THEN
                  KOUNT=KOUNT+1
                  EC(KOUNT)=SUM
                  IF (LAST+1.EQ.KOL) THEN
                     LAST=KOL
                     NSC(NOSEG,3)=KOL
                  ELSE
                     NOSEG=NOSEG+1
                     LAST=KOL
                     NSC(NOSEG,1)=LOW
                     NSC(NOSEG,2)=KOL
                     NSC(NOSEG,3)=KOL
```

```
            ENDIF
            SUM=0.0
        ENDIF
        I=LOW0
        L=L0
        I2=NSA(I,2)
        IF ((NST(J,1).EQ.KOL).AND.(J2.GT.NST(J,2))) THEN
            K=K+NST(J,3)-J2+1
            J=J+1
        ENDIF
        DO WHILE (NST(J,1).EQ.KOL)
            K=K+NST(J,3)-NST(J,2)+1
            J=J+1
        ENDDO
        J2=NST(J,2)
        KOL=NST(J,1)
    ELSEIF (NST(J,1).NE.KOL) THEN
        IF (SUM.NE.0.0) THEN
            KOUNT=KOUNT+1
            EC(KOUNT)=SUM
            IF (LAST+1.EQ.KOL) THEN
                LAST=KOL
                NSC(NOSEG,3)=KOL
            ELSE
                NOSEG=NOSEG+1
                LAST=KOL
                NSC(NOSEG,1)=LOW
                NSC(NOSEG,2)=KOL
                NSC(NOSEG,3)=KOL
            ENDIF
            SUM=0.0
        ENDIF
        I=LOW0
        L=L0
        I2=NSA(I,2)
        KOL=NST(J,1)
    ELSE
        IF (I2.GT.J2) THEN
            IF (I2.LE.NST(J,3)) THEN
                K=K+I2-J2
                IF (NSA(I,3).GT.NST(J,3)) THEN
                    DO M=I2,NST(J,3)
                        SUM=SUM+EA(L)*ET(K)
                        L=L+1
                        K=K+1
                    ENDDO
                    J=J+1
                    I2=M
                    J2=NST(J,2)
                ELSEIF (NSA(I,3).EQ.NST(J,3)) THEN
                    DO M=I2,NSA(I,3)
                        SUM=SUM+EA(L)*ET(K)
                        L=L+1
                        K=K+1
                    ENDDO
```

```
            I=I+1
            J=J+1
            I2=NSA(I,2)
            J2=NST(J,2)
        ELSE
            DO M=I2,NSA(I,3)
                SUM=SUM+EA(L)*ET(K)
                L=L+1
                K=K+1
            ENDDO
            J2=M
            I=I+1
            I2=NSA(I,2)
        ENDIF
    ELSE
        IF (J2.GT.NST(J,2)) THEN
            K=K+NST(J,3)-J2+1
        ELSE
            K=K+NST(J,3)-NST(J,2)+1
        ENDIF
        J=J+1
        J2=NST(J,2)
    ENDIF
ELSEIF (NSA(I,3).GE.J2) THEN
    L=L+J2-I2
    IF (NSA(I,3).GT.NST(J,3)) THEN
        DO M=J2,NST(J,3)
            SUM=SUM+EA(L)*ET(K)
            L=L+1
            K=K+1
        ENDDO
        I2=M
        J=J+1
        J2=NST(J,2)
    ELSEIF (NSA(I,3).EQ.NST(J,3)) THEN
        DO M=J2,NST(J,3)
            SUM=SUM+EA(L)*ET(K)
            L=L+1
            K=K+1
        ENDDO
        I=I+1
        J=J+1
        I2=NSA(I,2)
        J2=NST(J,2)
    ELSE
        DO M=J2,NSA(I,3)
            SUM=SUM+EA(L)*ET(K)
            L=L+1
            K=K+1
        ENDDO
        J2=M
        I=I+1
        I2=NSA(I,2)
    ENDIF
ELSE
```

```
                  IF (I2.GT.NSA(I,2)) THEN
                     L=L+NSA(I,3)-I2+1
                  ELSE
                     L=L+NSA(I,3)-NSA(I,2)+1
                  ENDIF
                  I=I+1
                  I2=NSA(I,2)
               ENDIF
            ENDIF
         ENDDO
         IF ((NSA(I,1).EQ.LOW).AND.(I2.GT.NSA(I,2))) THEN
            L=L+NSA(I,3)-I2+1
            I=I+1
         ENDIF
         DO WHILE (NSA(I,1).EQ.LOW)
            L=L+NSA(I,3)-NSA(I,2)+1
            I=I+1
         ENDDO
         LOW0=I
         L0=L
         LOW=NSA(I,1)
         I2=NSA(I,2)
      ENDDO
      NSC(0,1)=NSA(0,1)
      NSC(0,2)=NSB(0,2)
      NSC(0,3)=NOSEG
      RETURN
      END
C -----------------------------------------------------------------
C
C                              ******************
C                              *** TRANSPOSE ***
C                              ******************
C
C The subroutine TRANSPOSE generates the transpose of a matrix A given
C in the new representation, namely, (EA,NSA). The results is (EB,NSB).
C
C -----------------------------------------------------------------
C
      SUBROUTINE TRANSPOSE(EA,NSA,EB,NSB,QUE,KUE,NPNT,IDIM,JDIM,NSPO)
      DIMENSION EA(*),EB(*),NSA(0:NSPO,3),NSB(0:NSPO,3)
      DIMENSION QUE(JDIM,IDIM),KUE(JDIM,IDIM),NPNT(JDIM)
C
C ... construct the queue
C
      DO I=1,NSA(0,2)
         NPNT(I)=1
      ENDDO
      I=1
      KOUNT=0
      DO WHILE (I.LE.NSA(0,3))
         DO J=NSA(I,2),NSA(I,3)
            KOUNT=KOUNT+1
            KUE(J,NPNT(J))=NSA(I,1)
            QUE(J,NPNT(J))=EA(KOUNT)
```

```
            NPNT(J)=NPNT(J)+1
         ENDDO
         I=I+1
      ENDDO
C
C ... generate the transpose
C
      J=1
      K=1
      L=1
      KBLK=0
      LAST=-1
      DO WHILE (L.LE.KOUNT)
         IF (KUE(J,K).NE.0) THEN
            EB(L)=QUE(J,K)
            IF (LAST+1.EQ.KUE(J,K)) THEN
               LAST=LAST+1
               NSB(KBLK,3)=LAST
            ELSE
               LAST=KUE(J,K)
               KBLK=KBLK+1
               NSB(KBLK,1)=J
               NSB(KBLK,2)=LAST
               NSB(KBLK,3)=LAST
            ENDIF
            K=K+1
            L=L+1
         ELSE
            K=1
            J=J+1
            LAST=-1
         ENDIF
      ENDDO
      NSB(0,1)=NSA(0,2)
      NSB(0,2)=NSA(0,1)
      NSB(0,3)=KBLK
      RETURN
      END
C ---------------------------------------------------------------------
C
C                              ****************
C                              *** PACKING ***
C                              ****************
C
C The subroutine PACKING converts a standard two-dimensional matrix
C A into its corresponding sparse representation, namely, (EA,NSA).
C
C ---------------------------------------------------------------------
C   A(*,*)    :  Two-dimensional array representation of the matrix A.
C  (EA,NSA)   :  Sparse matrix representation of A, see routine MMULT.
C ---------------------------------------------------------------------
C
      SUBROUTINE PACKING(A,EA,NSA,IDIM,JDIM,NSPO)
      DIMENSION A(IDIM,JDIM),EA(*),NSA(0:NSPO,3)
      KOUNT=0
```

```
      NOSEG=0
      DO I=1,IDIM
         LAST=-1
         DO J=1,JDIM
            IF (A(I,J).NE.0.0) THEN
               KOUNT=KOUNT+1
               EA(KOUNT)=A(I,J)
               IF (LAST+1.EQ.J) THEN
                  LAST=J
                  NSA(NOSEG,3)=J
               ELSE
                  LAST=J
                  NOSEG=NOSEG+1
                  NSA(NOSEG,1)=I
                  NSA(NOSEG,2)=J
                  NSA(NOSEG,3)=J
               ENDIF
            ENDIF
         ENDDO
      ENDDO
      NSA(0,1)=IDIM
      NSA(0,2)=JDIM
      NSA(0,3)=NOSEG
      RETURN
      END
C -----------------------------------------------------------------
C
C                              ******************
C                              *** UNPACKING ***
C                              ******************
C
C The subroutine UNPACKING converts a sparse matrix representation
C (EA,NSA) of A back into its two-dimensional array representation.
C
C -----------------------------------------------------------------
C  (EA,NSA)   :  Sparse matrix representation of A, see routine MMULT.
C   A(*,*)    :  Two-dimensional array representation of the matrix A.
C -----------------------------------------------------------------
C
      SUBROUTINE UNPACKING(EA,NSA,A,IDIM,JDIM,NSPO)
      DIMENSION EA(*),A(IDIM,JDIM),NSA(0:NSPO,3)
      KOUNT=1
      DO I=1,NSA(0,3)
         DO J=NSA(I,2),NSA(I,3)
            A(NSA(I,1),J)=EA(KOUNT)
            KOUNT=KOUNT+1
         ENDDO
      ENDDO
      RETURN
      END
C -----------------------------------------------------------------
C
C                              *****************
C                              *** PRINTOUT ***
C                              *****************
```

```
C
C The subroutine PRINTOUT prints elements of a two-dimensional matrix.
C
C -----------------------------------------------------------------
C       A    : Two-dimensional array representation of the matrix A.
C     IDIM1  : First position of the row that is to be print out.
C     IDIM2  : Second position of the row that is to be print out.
C     JDIM1  : First position of the column that is to be print out.
C     JDIM1  : Second position of the column that is to be print out.
C -----------------------------------------------------------------
C
      SUBROUTINE PRINTOUT(A,IDIM,JDIM,IDIM1,IDIM2,JDIM1,JDIM2)
      DIMENSION A(IDIM,JDIM)
      JDIFF=JDIM2-JDIM1
      IF (JDIFF.LT.0) THEN
         WRITE(6,*)
     *    ' PRINT MESSAGE : JDIM2 TO BE LESS THAN OR EQUAL TO JDIM1'
         RETURN
      ELSEIF (JDIFF.GE.10) THEN
         WRITE(6,*)
     *    ' PRINT WARNING : ONLY FIRST 10 OF NUMBER REQUESTED GIVEN'
         JDIM2=JDIM1+9
      ENDIF
      DO I=IDIM1,IDIM2
         WRITE(6,100) I,(A(I,J),J=JDIM1,JDIM2)
      ENDDO
      RETURN
100   FORMAT(1X,I7,2X,10F7.1)
      END
```

CURRICULUM VITAE

Born: 02/27/1954

Education:

December 1991: **Ph.D.** (Computer Science), Louisiana State University, Louisiana, USA

August 1986: **M.Sc.** (Physics), Hampton University, Virginia, USA.

February 1982: **M.Eng.** (Applied Physics), Inha University, Incheon, Korea.

February 1979: **B.Eng.** (Applied Physics), Inha University, Incheon, Korea.

Publications:

J. P. Draayer, O. Castaños and S. C. Park, "Shell Model Interpretation of Collective Model Phenomenology," *International Conference on Contemporary Topics in Nuclear Structure Physics*, 345-373, 1988.

J. P. Draayer, Y. Leschber, S. C. Park and R. Lopez, "Representations of U(3) in U(N)," *Comp. Phys. Comm.*, **56**, 279-290, 1989.

J. P. Draayer, S. C. Park and O. Castaños, "Shell-Model Interpretation of the Collective-Model Potential-Energy Surface," *Physical Review Letters*, **62**, 20, 1989.

P. Rochford, S. C. Park, J. P. Draayer and S.-Q. Zheng, "Optimal Methods For Large-Scale Scientific Database and Sparse Matrix Applications," to be appeared in the AIP Proceedings of the Conference on Computational Quantum Physics held May 22-25, 1991, at Vanderbilt University, Nashville, TN.

S. C. Park and W. W. Buck, "Structure Functions for Electron-Necleon Coincidence," *Report of The 1985 Summer Study Group at CEBAF*, 8-24, 1985.

S. C. Park and J. P. Draayer, "Balanced Binary Tree Code for Scientific Applications," *Comp. Phys. Comm.*, **55**, 189, 1989.

S. C. Park, J. P. Draayer, and S. -Q. Zheng, "Priority Balanced Binary Tree for Large-Scale Scientific Applications," Technical Report #90-015, Department of Computer Science, Louisiana State University, 1990.

S. C. Park, J. P. Draayer, and S. -Q. Zheng, "Time-Space Optimal Numerical Database using Weighted Search Trees," Technical Report #90-014, Department of Computer Science, Louisiana State University, 1990.

S. C. Park, J. P. Draayer, and S.-Q. Zheng, "Time-Space Optimal Numerical Database for Large-Scale Scientific Applications," in Proceedings of the International Computer Symposium, December 17-19, 1990, Hsinchu, Taiwan, R.O.C.

S. C. Park, J. P. Draayer, and S. -Q. Zheng, "Numerical Database System Based on a Weighted Search Tree," submitted to *Comp. Phys. Comm.*.

S. C. Park, J. P. Draayer, and S. -Q. Zheng, "An Efficient Algorithm for Sparse Matrix Computations," Technical Report #91-009, Department of Computer Science, Louisiana State University, 1991 and also submitted to SAC '92.

Languages: English and Korean

DOCTORAL EXAMINATION AND DISSERTATION REPORT

Candidate: Soon Park

Major Field: Computer Science

Title of Dissertation: Efficient Data Structures and Algorithms for Scientific Computations.

Approved:

Major Professor and Chairman

Dean of the Graduate School

EXAMINING COMMITTEE:

Date of Examination:

August 21, 1991